Artificial Intelligence 241 (2016) 170–190

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

A parametric propagator for pairs of SUM constraints with a discrete convexity property ☆

Jean-Noël Monette [a,*], Nicolas Beldiceanu [b,**], Pierre Flener [a,**], Justin Pearson [a,**]

[a] Uppsala University, Dept. of Information Technology, 751 05 Uppsala, Sweden
[b] Mines Nantes, TASC (CNRS/INRIA), 44307 Nantes, France

ARTICLE INFO

Article history:
Received 17 June 2015
Received in revised form 25 August 2016
Accepted 31 August 2016
Available online 14 September 2016

Keywords:
Constraint programming
Propagator
Discrete convexity

ABSTRACT

We introduce a propagator for pairs of SUM constraints, where the expressions in the sums respect a form of convexity. This propagator is parametric and can be instantiated for various concrete pairs, including DEVIATION, SPREAD, and the conjunction of LINEAR$_{\leq}$ and AMONG. We show that despite its generality, our propagator is competitive in theory and practice with state-of-the-art propagators.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Constraint programming (CP) is a set of techniques to model and solve combinatorial problems using a mix of inference and search. Each constraint of the problem is associated with a *propagator* that performs pruning, i.e., the removal from the search space of impossible values for the variables of the constraint.

Many combinatorial problems involve SUM constraints, along with other constraints. It is however well-known that a propagator for a SUM constraint taken in isolation is not able to perform a lot of pruning since the estimation of the minimum or maximum of a sum does not take other constraints into account. Several authors have studied how to include other constraints (sharing some variables) in the propagator for SUM, either in particular cases (e.g., SPREAD [2], DEVIATION [3], INCREASINGSUM [4], and SUM with cliques [5]), or in general (e.g., OBJECTIVESUM [6]).

In the present work, we focus on a parametric constraint, called TWOSUMS hereafter, which can be cast as

$$\sum_{i\in[1,n]} f_i(x_i) \leq \overline{f} \tag{1}$$

$$\underline{g} \leq \sum_{i\in[1,n]} g_i(x_i) \leq \overline{g} \tag{2}$$

☆ This work was supported by grants 2011-6133 and 2012-4908 of the Swedish Research Council (VR). This paper is an invited revision of a paper which first appeared at the 18th International Conference on Principles and Practice of Constraint Programming (CP 2013). We thank the reviewers of CP 2013 for their constructive comments on the prior version [1] of this paper; the reviewers of this journal also greatly helped us to improve this paper.

* Principal corresponding author.
** Corresponding author.
E-mail addresses: Jean-Noel.Monette@tacton.com (J.-N. Monette), Nicolas.Beldiceanu@mines-nantes.fr (N. Beldiceanu), Pierre.Flener@it.uu.se (P. Flener), Justin.Pearson@it.uu.se (J. Pearson).

http://dx.doi.org/10.1016/j.artint.2016.08.006
0004-3702/© 2016 Elsevier B.V. All rights reserved.

Table 1
Several instantiations of the TwoSums constraint. For each constraint, we give the consistency achieved by the propagator presented in this paper along with its time complexity, as well as the complexity of previously published propagators achieving the same consistency. Here n is the number of variables, d is the size of the largest domain, and $d_\cup$ is the size of the union of the domains. The constraint definitions are given in Section 2.

Name	Consistency	Complexity	Specialised propagator
Linear$_\leq$	domain	$\mathcal{O}(n)$	$\mathcal{O}(n)$ [10]
Linear$_\leq$ $\wedge$ Among	domain	$\mathcal{O}(n \cdot (\log n + d))$	$\mathcal{O}(n \cdot (\log n + d))$ [9]
Linear$_\leq$ $\wedge$ Maximum	domain	$\mathcal{O}(n \cdot (\log n + d))$	–
Deviation	bounds($\mathbb{Z}$)	$\mathcal{O}(n)$	$\mathcal{O}(n)$ [3]
Spread	bounds($\mathbb{Z}$)	$\mathcal{O}(n \cdot d_\cup)$	$\mathcal{O}(n \log n)$ [11]
L_p-Norm, $0 < p < +\infty$	bounds($\mathbb{Z}$)	$\mathcal{O}(n \cdot d_\cup)$	–
Linear$_=$	bounds($\mathbb{R}$)	$\mathcal{O}(n)$	$\mathcal{O}(n)$ [10]
Linear$_\leq$ $\wedge$ Linear$_=$	bounds($\mathbb{R}$)	$\mathcal{O}(n^2)$	–

for any $n \geq 1$. The parameters f_i and g_i are functions from integers to integers and the f_i (respectively g_i) can differ for each i. Initially, we require $\overline{f}$, $\underline{g}$, and $\overline{g}$ to be constants, but Section 7 shows how to use variables instead, along with other extensions of this constraint. As usual when describing a constraint and its propagator, we assume that all x_i are distinct variables.

Finding a solution to the conjunction of (1) and (2) is in general NP-complete as it includes as a special case the knapsack problem (by taking $f_i(v) = a_i \cdot v$ and $g_i(v) = b_i \cdot v$, where the a_i and b_i are constants). In this paper, we will discuss a large class of functions f_i and g_i for which either domain consistency, bounds($\mathbb{Z}$) consistency, or bounds($\mathbb{R}$) consistency [7,8] can be achieved in polynomial time. We present a *parametric propagator* for this tractable class and show how to instantiate it for various functions f_i and g_i. We show that the instantiations include among others bounds($\mathbb{Z}$)-consistent propagators for the constraints Deviation [3] and Spread [2] and a domain-consistent propagator for the conjunction of Linear$_\leq$ and Among [9].

While the worst-case time complexity of our parametric propagator is $\mathcal{O}(n \cdot d^2 + n^2 \cdot d)$, where d is the size of the largest domain,[1] our propagator, once instantiated, matches the theoretical time complexity and practical efficiency of several previously published specialised propagators, as shown in Table 1. However, our propagator is not limited to the reproduction of existing propagators. Table 1 also lists the time complexity and consistency for several instantiations that we identified as being useful general cases but for which, to the best of our knowledge, no propagator existed. This list is not exhaustive and one can add many problem-specific instantiations: see for instance Example 3. Note that while achieving domain consistency on the knapsack problem is NP-hard, bounds($\mathbb{R}$) consistency is achieved in polynomial time by our propagator (last line in Table 1).

After introducing some notation and background in Section 2, we present our TwoSums propagator in Sections 3 to 5. We study its complexity in Section 6 and give some implementation notes. In Section 7, we show how the applicability of the propagator can be extended. Afterwards, we present in Section 8 several instantiations of the propagator, including a detailed case study for Deviation. Finally, Section 9 presents some experimental results showing that the genericity of our approach is not detrimental to performance. We review the related work in Section 10 before concluding in Section 11.

This paper is an extended version of [1], expanded with new examples, proofs, and implementation notes.

2. Notation and background

For a function f and value v, we write $f^{-1}(v)$ for the *set* of values having v as image: $\{u \mid f(u) = v\}$. For a function f and set S, we write $f(S)$ for the set of images of the elements of S: $\{f(u) \mid u \in S\}$. We use x_i, v_i, f_i to represent single variables, values, and functions, while $\mathbf{x}$, $\mathbf{v}$, $\mathbf{f}$ represent vectors of variables, values, and functions (e.g., $\mathbf{x} = \langle x_1, x_2, \ldots, x_n \rangle$).

Constraint programming (CP) is a set of techniques to model and solve problems defined by a set of existentially quantified variables and a set of constraints over those variables. During the solution process, each variable x is associated with a current *domain*, denoted D_x, of candidate values for this variable. Each variable is given an initial domain in the model. The smallest and largest values of D_x are denoted respectively $\min(\mathrm{D}_x)$ and $\max(\mathrm{D}_x)$. For a constraint c and the variables $\mathbf{x}$ involved in c, an assignment of values $v_i \in \mathrm{D}_{x_i}$, for all $x_i \in \mathbf{x}$, is called a *solution* to c if the vector $\mathbf{v}$ of values satisfies c.

One of the main operations in CP-style constraint solving is the *filtering* of the domain of a variable with respect to a constraint: values that can be proven not to belong to any solution of a constraint c (given the domains of the other variables) are removed from the domain of each variable. This is performed by an algorithm called a *propagator* of c.

A propagator does not need to remove all values that do not participate in any solution, as doing so can be computationally too expensive. If a propagator removes all such values, then it is said to enforce *domain consistency*. If a propagator always enforces domain consistency, then it is called *domain consistent*. Weaker notions of consistency exist. In particular, *bounds*($\mathbb{Z}$) *consistency* (see, e.g., [7,8]) is achieved if, for each variable x, both $\min(\mathrm{D}_x)$ and $\max(\mathrm{D}_x)$ belong to a solution in which the other variables take supporting integer values within their domain bounds. If a propagator always enforces

[1] The precise expression of the time complexity also depends on several other parameters introduced later and is detailed in Section 6.

bounds($\mathbb{Z}$) consistency, then it is called *bounds*($\mathbb{Z}$) *consistent*. Similarly, *bounds*($\mathbb{R}$) *consistency* (see also, e.g., [7,8]) is achieved if, for each variable x, both $\min(D_x)$ and $\max(D_x)$ belong to a solution in which the other variables take supporting real values within their domain bounds. Bounds($\mathbb{R}$) consistency is weaker than bounds($\mathbb{Z}$) consistency in that it considers that supporting values need not be integers. Bounds($\mathbb{R}$) consistency is the consistency usually considered for the implementation of propagators for LINEAR constraints. If a propagator always enforces bounds($\mathbb{R}$) consistency, then it is called *bounds*($\mathbb{R}$) *consistent*.

In this work, we make the distinction between the SUM constraint and the LINEAR constraints. A SUM constraint is any constraint $\sum_{i\in[1,n]} f_i(x_i)\ \#\ y$, for any functions f_i from integers to integers, with $\# \in \{<, \leq, =, \neq, \geq, >\}$, and y being an integer variable or constant. The f_i in the SUM constraint may be instantiated to give rise to many existing constraints. In particular, $\text{LINEAR}_{\leq}(\mathbf{x}, \mathbf{a}, s)$ holds if and only if the weighted sum of variables x_i with given integer weights a_i is at most the integer variable s, i.e., $\sum_{i\in[1,n]} a_i \cdot x_i \leq s$. Similarly, $\text{LINEAR}_{=}(\mathbf{x}, \mathbf{a}, s)$ holds if and only if the weighted sum of variables x_i with given integer weights a_i is equal to the integer variable s, i.e., $\sum_{i\in[1,n]} a_i \cdot x_i = s$. These constraints have been studied among others in [10] where many practical improvements of the usual $\mathcal{O}(n)$ bounds($\mathbb{R}$)-consistent propagator are introduced. In the case of $\text{LINEAR}_{\leq}$, bounds($\mathbb{R}$) consistency coincides with domain consistency.

The $\text{SPREAD}(\mathbf{x}, \mu, s)$ constraint [2] holds if and only if the average of the integer variables x_i is the given rational number μ and the sum of their scaled squared deviations from μ is less than or equal to the integer variable s, i.e., $\sum_{i\in[1,n]} x_i = n \cdot \mu \wedge \sum_{i\in[1,n]} (n \cdot x_i - n \cdot \mu)^2 \leq s$. While a generalised constraint with a variable average exists [2], we consider here only the case of a fixed average, which is used in most applications. Following [2], all values in the second inequality are scaled by n to work with integer values, as in general the average μ might not be integer but $n \cdot \mu$ surely is. An $\mathcal{O}(n \cdot \log n)$ bounds($\mathbb{Z}$)-consistent propagator for SPREAD has been introduced in [11].

Similarly, the $\text{DEVIATION}(\mathbf{x}, \mu, d)$ constraint [3] holds if and only if the average of the integer variables x_i is the given rational number μ and the sum of their scaled deviations from μ is less than or equal to the integer variable d, i.e., $\sum_{i\in[1,n]} x_i = n \cdot \mu \wedge \sum_{i\in[1,n]} |n \cdot x_i - n \cdot \mu| \leq d$. Again, we consider only the case of a fixed average. An $\mathcal{O}(n)$ bounds($\mathbb{Z}$)-consistent propagator for DEVIATION has been presented in [3].

The constraint $\text{L}_p\text{-NORM}(\mathbf{x}, \mu, d)$, with $0 < p < +\infty$ holds if and only if $\sum_{i\in[1,n]} x_i = n \cdot \mu \wedge \sum_{i\in[1,n]} |n \cdot x_i - n \cdot \mu|^p \leq s$. It generalises SPREAD (with $p = 2$) and Deviation (with $p = 1$).

The $\text{AMONG}(\mathbf{x}, \mathcal{V}, c)$ constraint holds if and only if the number of integer variables x_i taking their value in the given integer set $\mathcal{V}$ is equal to the integer variable c. The AMONG constraint can be represented using a SUM constraint as $\sum_{i\in[1,n]} [x_i \in \mathcal{V}] = c$, where the notation $[\gamma]$ is the Iverson bracket and is defined to be 1 if γ is true, and 0 otherwise. A domain-consistent propagator for the conjunction of a $\text{LINEAR}_{\leq}$ constraint and an AMONG constraint has been published in [9], with an $\mathcal{O}(n \cdot (\log n + d))$ time complexity, where d is the size of the largest D_{x_i}.

3. Overview of the approach

Our approach for propagating the TWOSUMS constraint contains two parts. First (as discussed in Section 4), we compute a sharp lower bound b on the left-hand side $\sum_{i\in[1,n]} f_i(x_i)$ of constraint (1) under constraint (2), together with a *support* $\mathbf{s}^b$, i.e., an assignment of the x_i yielding value b and satisfying constraint (2). The conjunction of (1) and (2) is feasible if and only if this lower bound, which we call the *feasibility bound*, is at most $\overline{f}$, the right-hand side of constraint (1). To compute this feasibility bound, we introduce new functions derived from the f_i and g_i. We show that the feasibility bound can be greedily computed if the newly introduced functions are *discretely convex* [12].

Example 1. For the conjunction of $\text{LINEAR}_{\leq}(\mathbf{x}, \mathbf{a}, s)$ and $\text{AMONG}(\mathbf{x}, \mathcal{V}, c)$ [9], the feasibility bound is computed by first taking for each variable an extreme value in its domain, and then iteratively modifying the values taken by some variables until the AMONG constraint is satisfied. The initial value is the largest in the domain of x_i if $a_i < 0$ and is the smallest otherwise, so that the value of the weighted sum is as small as possible. The iterative modification picks a variable such that changing its value causes the smallest increase to the bound. As the increase caused by a variable is independent of the values taken by the other variables, the iteration order can be determined beforehand by sorting. Full details are provided in [9]. □

In the second part (discussed in Section 5) of our propagator for TWOSUMS, the domain of each variable x_j is filtered by computing for each value u in its domain a sharp lower bound on the left hand side $\sum_{i\in[1,n]} f_i(x_i)$ of constraint (1) under constraint (2) when x_j is assigned u. If this lower bound is larger than $\overline{f}$, then u is removed from the domain of x_j. The lower bound for each pair (x_j, u) is computed *incrementally* from the support $\mathbf{s}^b$ for the feasibility bound, thanks to the discrete convexity property, and does not need to be computed explicitly for each value. We also present an improved propagator when an additional property holds on f_j and g_j, namely a form of monotonicity.

Example 2. For the conjunction of $\text{LINEAR}_{\leq}(\mathbf{x}, \mathbf{a}, s)$ and $\text{AMONG}(\mathbf{x}, \mathcal{V}, c)$, the filtering for each variable x_j amounts to computing two values: the lower bound $b_j^{\in}$ for the weighted sum of all the other variables when variable x_j is assigned a value in $\mathcal{V}$, and the lower bound $b_j^{\notin}$ when x_j is assigned a value not in $\mathcal{V}$. One of these two bounds ($b_j^{\in}$ if $s_j^b \in \mathcal{V}$ and $b_j^{\notin}$ otherwise) is directly computed from the feasibility bound b by subtracting the contribution of x_j. The other bound can be computed incrementally from the first bound by modifying the value of another variable than x_j so that the modification

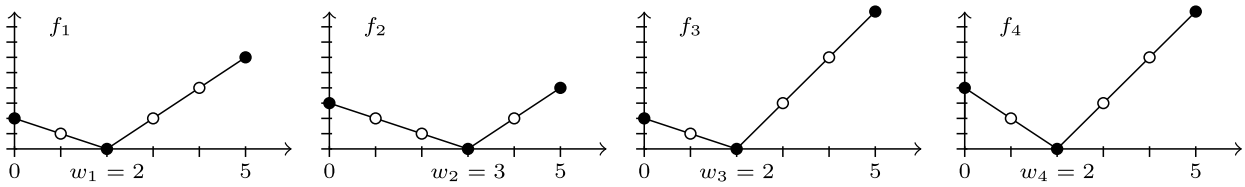

Fig. 1. The f_i functions of Example 3.

causes the smallest increase of the feasibility bound. The two new bounds are then used to remove values from the domain of x_j: the bound $b_j^{\in}$ is used to remove possibly values that are in $\mathcal{V}$, and $b_j^{\notin}$ to remove possibly values that are not in $\mathcal{V}$. Again, details are provided in [9]. □

Our propagator for TwoSums is parametric, depending on the f_i and g_i. The time complexity and the level of consistency achieved depend on the shape of the f_i and g_i and on the values given to the parameters (see Sections 6 and 7).

Example 3. Throughout the rest of the paper, we will use the following running example. Consider a workshop with n workers. The unknown daily workload in number of hours of each worker is encoded by variable x_i for each worker $i \in [1, n]$. The daily total workload must be equal to a given integer t. In addition, each worker has a given nominal workload w_i, from which they can deviate but at some cost given by the following function: $f_i(v) = \max\{r_i \cdot (w_i - v), q_i \cdot (v - w_i)\}$, i.e., the cost increases linearly with a slope of $r_i > 0$ if the actual workload is below the nominal one, and with a slope of $q_i > 0$ if the actual workload is above the nominal one. The total cost must be under some given integer upper bound c. This part of the problem (a typical workshop would have additional constraints) is modelled as:

$$\sum_{i \in [1,n]} \max\{r_i \cdot (w_i - x_i), q_i \cdot (x_i - w_i)\} \leq c$$

$$t \leq \sum_{i \in [1,n]} x_i \leq t$$

This is an instantiation of constraints (1) and (2). In subsequent examples, we will use the following values for the parameters: $n = 4$, $c = 5$, $t = 10$, $\mathbf{w} = \langle 2, 3, 2, 2 \rangle$, $\mathbf{r} = \langle 1, 1, 1, 2 \rangle$, and $\mathbf{q} = \langle 2, 2, 3, 3 \rangle$. We also define $\mathrm{D}_{x_i} = [0, 5]$ for all $i \in [1, 4]$. Fig. 1 shows the functions f_i over the given domains. □

4. Feasibility test

The TwoSums constraint is satisfiable if and only if the *cost* (i.e., the value of the objective function) of an optimal solution to the following problem is at most the value $\overline{f}$:

$$\begin{aligned} \text{minimise} \quad & \sum_{i \in [1,n]} f_i(x_i) \\ \text{such that} \quad & \underline{g} \leq \sum_{i \in [1,n]} g_i(x_i) \leq \overline{g} \\ & x_i \in \mathrm{D}_{x_i}, \quad \forall i \in [1, n] \end{aligned} \tag{3}$$

We gradually show in the next sub-sections how to compute greedily this cost, called the *feasibility bound*, together with a support, which will be used for filtering in Section 5.

4.1. Problem reformulation

We reformulate problem (3) into a simpler problem in two steps. The first step introduces for each i a new function h_i that captures the relation between f_i and g_i. This results in having only one function per variable instead of two. The second step splits the resulting reformulated problem into two subproblems that can be solved separately.

First step After introducing new variables y_i, so that $y_i = g_i(x_i)$ for each i, we state the following new problem:

$$\begin{aligned} \text{minimise} \quad & \sum_{i \in [1,n]} h_i(y_i) \\ \text{such that} \quad & \underline{g} \leq \sum_{i \in [1,n]} y_i \leq \overline{g} \\ & y_i \in g_i(\mathrm{D}_{x_i}), \quad \forall i \in [1, n] \end{aligned} \tag{4}$$

where we introduce a new function $h_i\colon g_i(\mathrm{D}_{x_i}) \to f_i(\mathrm{D}_{x_i})$ for each i. This function is defined as $h_i(v) = \min f_i(g_i^{-1}(v)) = \min\{f_i(u) \mid u \in \mathrm{D}_{x_i} \wedge g_i(u) = v\}$, that is $h_i(v)$ is the smallest value of $f_i(x_i)$ that can be attained when $g_i(x_i)$ is equal to value v. Note that the definition of h_i depends on the *current* domain of x_i.

Example 4. In Example 3, we have $g_i(u) = u$ for all u, hence $g_i^{-1}(u) = \{u\}$. It follows that $h_i(v) = f_i(v)$, but restricted to the current domain of x_i. □

We now prove that the feasibility bound can also be computed from problem (4).

Lemma 1. *All optimal solutions to problems (3) and (4) have the same cost.*

Proof. Let $\mathbf{v}$ denote a vector of values for the vector $\mathbf{y}$ of variables. For each value v_i, we choose an arbitrary value u_i in D_{x_i} such that $g_i(u_i) = v_i$ and $f_i(u_i) = h_i(v_i)$. Such a value u_i always exists, by the definition of h_i. Then the vector $\mathbf{u}$ is a feasible solution to problem (3) if and only if $\mathbf{v}$ is a feasible solution to problem (4), and they have the same cost. In addition, any other assignment $\mathbf{u}'$ such that $g_i(u_i') = v_i$ for each i has a cost larger than or equal to the cost of $\mathbf{u}$ and $\mathbf{v}$, by the definition of h_i. Hence $\mathbf{u}$ is optimal if and only if $\mathbf{v}$ is optimal. □

Second step We define a new function, called H, from integers to integers:

$$H(b) = \min\left\{ \sum_{i\in[1,n]} h_i(y_i) \;\middle|\; \sum_{i\in[1,n]} y_i = b \wedge \forall i \in [1,n] : y_i \in g_i(\mathrm{D}_{x_i}) \right\} \tag{5}$$

That is, $H(b)$ is the minimum of the sum of the $h_i(y_i)$ when the sum of the y_i is equal to b. For a given integer b, we define $\mathbf{s}^b$ to be an assignment to $\mathbf{y}$ such that $b = \sum_{i\in[1,n]} s_i^b$ and $H(b) = \sum_{i\in[1,n]} h_i(s_i^b)$. We call $\mathbf{s}^b$ a *support* for b. We propose the following new problem:

$$\begin{aligned} &\text{minimise} \quad H(z) \\ &\text{such that} \quad \underline{g} \le z \le \overline{g} \end{aligned} \tag{6}$$

where z is a new variable. We now prove that the feasibility bound can also be computed from problem (6), as the latter has the same optimal cost as problem (4), and as problem (3) by Lemma 1.

Lemma 2. *All optimal solutions to problems (4) and (6) have the same cost.*

Proof. This is shown by replacing $H(z)$ by its definition (5) in the formulation of problem (6). This gives

$$\min\left\{ \min\left\{ \sum_{i\in[1,n]} h_i(y_i) \;\middle|\; \sum_{i\in[1,n]} y_i = z \wedge \forall i \in [1,n] : y_i \in g_i(\mathrm{D}_{x_i}) \right\} \;\middle|\; \underline{g} \le z \le \overline{g} \right\}$$

which is equal to

$$\min\left\{ \sum_{i\in[1,n]} h_i(y_i) \;\middle|\; \sum_{i\in[1,n]} y_i = z \wedge \forall i \in [1,n] : y_i \in g_i(\mathrm{D}_{x_i}) \wedge \underline{g} \le z \le \overline{g} \right\}$$

Substituting z by $\sum_{i\in[1,n]} y_i$ leads to the formulation of problem (4). □

Problems (4) and (6) are more interesting than problem (3) in three respects. First, it is simpler to reason with only one function per variable (namely h_i) instead of two (namely f_i and g_i). Second, the domain D_{y_i}, which is equal to $g_i(\mathrm{D}_{x_i})$, might be much smaller than D_{x_i} (but never larger), potentially reducing a lot the number of values the algorithms must consider. Third, introducing H allows us to compute the feasibility bound in two steps: (*i*) construct H from the h_i, and (*ii*) find an optimal solution to (6). This can be done greedily, as we will show in Section 4.4, if all h_i are discretely convex, which is a concept we recall now.

Definition 1 *([12]).* A function $f\colon A \to B$, where $A, B \subseteq \mathbb{Z}$, is *discretely convex* if

1. A is an interval, and
2. $\forall v \in A\colon (v-1) \in A \wedge (v+1) \in A \Rightarrow 2 \cdot f(v) \le f(v-1) + f(v+1)$.

Table 2
Several instantiations of f_i and g_i, and the corresponding h_i. The notation $[\gamma]$ is the Iverson bracket and is defined to be 1 if γ is true, and 0 otherwise.

Name	$f_i(u)$	$g_i(u)$	$h_i(v)$
LINEAR$_\leq$ (alone)	$a_i \cdot u$	0	$\begin{cases} a_i \cdot \min \mathrm{D}_{x_i} & \text{if } a_i > 0 \\ a_i \cdot \max \mathrm{D}_{x_i} & \text{if } a_i \leq 0 \end{cases}$
LINEAR$_\leq$ $\wedge$ AMONG [9]	$a_i \cdot u$	$[u \in \mathcal{V}]$	$\begin{cases} a_i \cdot \min\left(\mathrm{D}_{x_i} \setminus \mathcal{V}\right) & \text{if } v = 0 \wedge a_i > 0 \\ a_i \cdot \max\left(\mathrm{D}_{x_i} \setminus \mathcal{V}\right) & \text{if } v = 0 \wedge a_i \leq 0 \\ a_i \cdot \min\left(\mathrm{D}_{x_i} \cap \mathcal{V}\right) & \text{if } v = 1 \wedge a_i > 0 \\ a_i \cdot \max\left(\mathrm{D}_{x_i} \cap \mathcal{V}\right) & \text{if } v = 1 \wedge a_i \leq 0 \end{cases}$
LINEAR$_\leq$ $\wedge$ MAXIMUM	$a_i \cdot u$	$[u \geq m]$	$\begin{cases} a_i \cdot \min \mathrm{D}_{x_i} & \text{if } v = 0 \wedge a_i > 0 \\ a_i \cdot \max\left\{u \mid u \in \mathrm{D}_{x_i} \wedge u < m\right\} & \text{if } v = 0 \wedge a_i \leq 0 \\ a_i \cdot \min\left\{u \mid u \in \mathrm{D}_{x_i} \wedge u \geq m\right\} & \text{if } v = 1 \wedge a_i > 0 \\ a_i \cdot \max \mathrm{D}_{x_i} & \text{if } v = 1 \wedge a_i \leq 0 \end{cases}$
DEVIATION [3]	$\lvert n \cdot u - n \cdot \mu \rvert$	u	$\lvert n \cdot v - n \cdot \mu \rvert$
SPREAD [2]	$(n \cdot u - n \cdot \mu)^2$	u	$(n \cdot v - n \cdot \mu)^2$
L_p-NORM, $0 < p < +\infty$	$\lvert n \cdot u - n \cdot \mu \rvert^p$	u	$\lvert n \cdot v - n \cdot \mu \rvert^p$
LINEAR$_=$	0	$b_i \cdot u$	0
LINEAR$_\leq$ $\wedge$ LINEAR$_=$	$a_i \cdot u$	$b_i \cdot u$	$\frac{a_i}{b_i} \cdot u$
MOD_AND_DIV ($a_i > 0$)	$u - a_i \cdot \lfloor u/a_i \rfloor$	$\lfloor u/a_i \rfloor$	$\max\left\{0, \min \mathrm{D}_{x_i} - a_i \cdot v\right\}$

The notion of discrete convexity is an adaptation of the usual convexity from the reals to the integers. The intuition is that a function on integers is discretely convex if its natural extension to the reals is convex. This notion has been studied in depth, for instance in [12]. It is also related to the notion of submodular functions on sets [13].

The two conditions in Definition 1 on the h_i theoretically restrict the applicability of our approach. Those restrictions and their lifting in practice are discussed further in Section 7.1 but the next example already shows the broad applicability of our approach.

Example 5. Table 2 presents the functions f_i, g_i, and h_i for several pairs of constraints.

- The first line shows the example of a constant g_i function, i.e., $g_i(u)$ does not depend on the value of u, in which case the domain of h_i is a singleton and h_i is trivially discretely convex.
- If g_i is a characteristic function, i.e., a function taking only values 0 and 1 as when defined using the Iverson bracket (e.g., LINEAR$_\leq$ and AMONG, as well as LINEAR$_\leq$ and MAXIMUM), then the domain of h_i is composed of only two values and h_i is typically defined by giving a formula for each value. In those cases, h_i is always discretely convex.
- If g_i is the identity function, as for DEVIATION, SPREAD, and L_p-NORM, then h_i is equal to f_i. In such a case, h_i is discretely convex as long as D_{x_i} is relaxed to its smallest enclosing interval. As discussed in Section 7.1, this relaxation maintains the correctness of our approach but only bounds($\mathbb{Z}$) consistency can be achieved (see Table 1).
- If g_i is a linear function, as for LINEAR$_=$ alone and the conjunction of LINEAR$_\leq$ and LINEAR$_=$, then h_i is not discretely convex because $g_i(\mathrm{D}_{x_i})$ might not be an interval even if D_{x_i} is an interval. However, as discussed in Section 7.1, relaxing the domain of h_i to its smallest enclosing interval maintains the correctness of our approach: as shown in Table 1, one achieves only bounds($\mathbb{R}$) consistency.
- The last line shows the example of a pair of constraints f_i and g_i that are very different from the previous pairs but still give rise to a discretely convex h_i when D_{x_i} is relaxed to its smallest enclosing interval. □

Before providing algorithms in Section 4.4 to compute the feasibility bound, we need to introduce some notions in Section 4.2 and characterise H in Section 4.3.

4.2. Deltas, segments, slopes, and breakpoints

Let $f : A \to B$ be an arbitrary function with $A, B \subseteq \mathbb{Z}$. Given some value v in A, we call *right delta* (respectively *left delta*) the increase of f when v increases (respectively decreases) by 1. Formally: $\Delta^+(f, v) = f(v+1) - f(v)$ and $\Delta^-(f, v) = f(v-1) - f(v)$; the value of $\Delta^+(f, v)$ (respectively $\Delta^-(f, v)$) is $+\infty$ when $v+1$ (respectively $v-1$) is not in A.

A *segment* of f is a maximal interval $[\ell, u]$ of its domain where the right delta is constant. Formally: $\Delta^+(f, v) = \Delta^+(f, v+1)$ for all $v \in [\ell, u-1]$, with $\ell \leq u$, $\Delta^+(f, \ell-1) \neq \Delta^+(f, \ell)$, and $\Delta^+(f, u-1) \neq \Delta^+(f, u)$. The endpoints ℓ and u of a segment $[\ell, u]$ of f are called *breakpoints* of f. The *length* of a segment $[\ell, u]$ is $u - \ell$, that is the length of a line from ℓ to u, and not the number $u - \ell + 1$ of its integer elements. The *slope* of a segment $[\ell, u]$ is $\Delta^+(f, \ell)$. Hence the slope of a function is constant inside any of its segments and changes at its breakpoints.

The domain of f can be uniquely partitioned into its segments, and each value of the domain belongs to one or two segments. For a value v, the *breakpoint on the right* of v, denoted by $\mathrm{bp}^+(f, v)$, is u if v belongs to some segment $[\ell, u]$

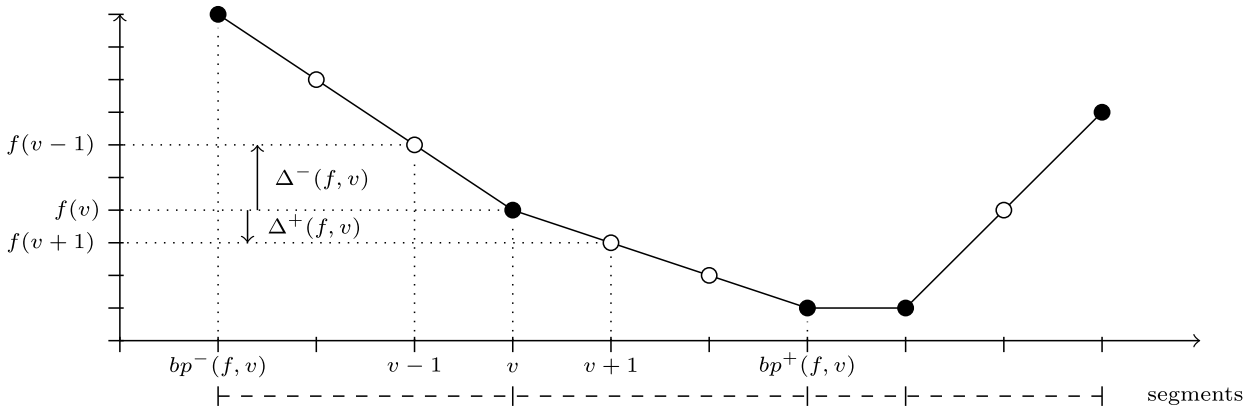

Fig. 2. Illustration of the notions of Section 4.2. Filled points appear at breakpoints.

with $u \neq v$, and otherwise undefined, denoted by $+\infty$. Similarly, $\mathrm{bp}^-(f, v)$ denotes the *breakpoint on the left* of v, if any, otherwise $-\infty$.

Let f be a discretely convex function. For any two consecutive segments, the slope of the former is smaller than the slope of the latter, hence no two segments have the same slope. Also, $\Delta^+(f, v) = +\infty$ only for the largest value v in A, because A is an interval, and $\Delta^-(f, v) = +\infty$ only for the smallest value v in A.

Fig. 2 illustrates these notions on a discretely convex function. The points are part of the actual function, while the segments joining them are used as a visual guide to identify the segments and their slopes.

Example 6. Each function h_i of Example 3 is composed of two segments (see Fig. 1). For $i = 1$, the function h_1, defined as $h_1(u) = \max\{1 \cdot (2 - u), 2 \cdot (u - 2)\}$ over the interval $[0, 5]$, is composed of the following two segments: one spans the interval $[0, 2]$ with slope -1, the other spans the interval $[2, 5]$ with slope 2. For $u \in [0, 1]$, we have $\Delta^+(h_1, u) = -1$ and $\mathrm{bp}^+(h_1, u) = 2$. For $u \in [2, 4]$, we have $\Delta^+(h_1, u) = 2$ and $\mathrm{bp}^+(h_1, u) = 5$. Finally, $\Delta^+(h_1, 5) = +\infty$ and $\mathrm{bp}^+(h_1, 5) = +\infty$. □

The basic properties of the special values $+\infty$ and $-\infty$ used in our algorithms are, for any $v \in \mathbb{Z}$: $-\infty < v < +\infty$, $v + (+\infty) = +\infty$, $v + (-\infty) = -\infty$, $v - (-\infty) = +\infty$, $v - (+\infty) = -\infty$, $\min(v, +\infty) = v$, and $v / +\infty = 0$.

4.3. Characterisation of the H function

We will show in Section 4.4 that when the h_i are discretely convex, problem (6) is easy to solve by greedy search, because H is then also discretely convex and can be calculated efficiently. In order to prove those claims, we first need to study closely the functions H and h_i, and the relationship between them and between supports. We first show how one can incrementally get a support for a value $b + 1$ from a support for a value b.

Lemma 3. *If each h_i is discretely convex, then given a support $\mathbf{s}^b$ for some value b, there exists a support $\mathbf{s}^{b+1}$ for $b + 1$ and some $j \in [1, n]$ such that $s_i^{b+1} = s_i^b$ for all $i \neq j$, and $s_j^{b+1} = s_j^b + 1$ (assuming b and $b + 1$ are in the domain of H).*

Proof. By definition, a support $\mathbf{s}^b$ for any b is such that $\sum_{i \in [1,n]} s_i^b = b$ and $\sum_{i \in [1,n]} h_i(s_i^b) = H(b)$. For any j and k with $k \neq j$, the sum

$$s_1^b + \cdots + (s_j^b + 1) + \cdots + (s_k^b - 1) + \cdots + s_n^b$$

also equals b as we added 1 to one value and removed 1 from another one.[2] Hence by definition of H (since the s_i^b are the values that minimise $H(b)$), we have:

$$H(b) \leq h_1(s_1^b) + \cdots + h_j(s_j^b + 1) + \cdots + h_k(s_k^b - 1) + \cdots + h_n(s_n^b)$$

Rearranging and cancelling out common terms gives:

$$h_k(s_k^b) - h_k(s_k^b - 1) \leq h_j(s_j^b + 1) - h_j(s_j^b) \tag{7}$$

[2] Visually, we here take $j < k$ but the reasoning does not depend on their order.

If h_j is discretely convex, then we have that:

$$h_k(s_k^b) - h_k(s_k^b - 1) \le h_j(s_j^b + 1) - h_j(s_j^b) \le h_j(s_j^b + 2) - h_j(s_j^b + 1)$$

And:

$$h_1(s_1^b) + \cdots + h_j(s_j^b + 1) + \cdots + h_k(s_k^b) + \cdots + h_n(s_n^b) \le h_1(s_1^b) + \cdots + h_j(s_j^b + 2) + \cdots + h_k(s_k^b - 1) + \cdots + h_n(s_n^b)$$

This means that adding two to any single s_j^b and reducing another s_k^b by one to arrive at the sum $b+1$ will have a higher cost than simply adding one to a single s_j^b. Because each h_i is discretely convex, this is true for any increment larger than one. Hence it is possible to find a support $\mathbf{s}^{b+1}$ for $b+1$ from a support $\mathbf{s}^b$ for b by increasing any suitable s_i^b by one. □

Lemma 3 also shows that it is possible to find a support $\mathbf{s}^{b-1}$ by subtracting one from any suitable s_i^b. We can now prove the following important result:

Theorem 4. *If each h_i is discretely convex, then H is discretely convex.*

Proof. The domain of each h_i is an interval $[s_i, u_i]$, so that the domain of H is the interval $\left[\sum_{i\in[1,n]} s_i, \sum_{i\in[1,n]} u_i\right]$. We need to show that $H(b) - H(b-1) \le H(b+1) - H(b)$. If s_i^b is a support for some b then by Lemma 3 there are some k and j such that $H(b-1) = h_1(s_1^b) + \cdots + h_k(s_k^b - 1) + \cdots + h_n(s_n^b)$ and $H(b+1) = h_1(s_1^b) + \cdots + h_j(s_j^b + 1) + \cdots + h_n(s_n^b)$. Therefore $H(b) - H(b-1) = h_k(s_k^b) - h_k(s_k^b - 1)$ and $H(b+1) - H(b) = h_j(s_j^b + 1) - h_j(s_j^b)$ and, by (7), $H(b) - H(b-1) \le H(b+1) - H(b)$. Hence H is discretely convex. □

We now show how to calculate H efficiently by giving a characterisation of its minimum and its segments. Here, for any non-empty set S and function f, the expression $\operatorname{argmin}_{i\in S} f(i)$ returns *one* arbitrary value $i \in S$ that minimises $f(i)$. In the following, b^* represents a value minimising the value of $H(b^*)$.

Lemma 5. *A support $\mathbf{s}^{b^*}$ for a value b^* that minimises H is such that $s_i^{b^*} = \operatorname{argmin}_{v_i \in g_i(\mathrm{D}_{x_i})} h_i(v_i)$ for all $i \in [1, n]$.*

Proof. If $\mathbf{s}^{b^*}$ is a support for b^*, then b^* is equal to $\sum_{i\in[1,n]} s_i^{b^*}$ and $H(b^*) = \sum_{i\in[1,n]} h_i(s_i^{b^*})$. Since each $s_i^{b^*} = \operatorname{argmin}_{v_i \in g_i(\mathrm{D}_{x_i})} h_i(v_i)$ corresponds to the minimum value obtainable by h_i, it is not possible to reduce the value $\sum_{i\in[1,n]} h_i(s_i^{b^*})$ by picking a different value for any $s_i^{b^*}$. □

There are potentially several $\mathbf{s}^{b^*}$ that minimise H. The correctness of our approach does not depend on a particular choice of support.

Example 7. For the constraints of Example 3, there is only one value minimising h_i for each i, namely the nominal workload w_i. Hence, the unique $\mathbf{s}^{b^*}$ is equal to the nominal workloads $\mathbf{w} = \langle 2, 3, 2, 2\rangle$. Then $b^* = 9 = 2 + 3 + 2 + 2$ and $H(b^*) = 0$ as $h_i(w_i) = 0$ for all i. □

We now characterise the segments of H. We first establish the relation between the slope of H at some value b and the slope of each h_i at s_i^b.

Lemma 6. *If $\mathbf{s}^b$ is a support for some value b, then $\Delta^+(h_i, s_i^b) \ge \Delta^+(H, b)$ and $\Delta^-(h_i, s_i^b) \ge \Delta^-(H, b)$ for all $i \in [1, n]$.*

Proof. If b is increased by one, then one of the s_i^b must be increased by one (by Lemma 3). To reach the minimum value for $b+1$, one needs to increase the value of a variable y_k for which $\Delta^+(h_k, s_k^b)$ is the smallest. So the increase of H, namely $\Delta^+(H, b)$, is equal to $\Delta^+(h_k, s_k^b)$, which is smaller than or equal to $\Delta^+(h_i, s_i^b)$ for any other i. A similar argument is used for a decrease of b. □

The proof of Lemma 6 also shows that there is at least one i such that $\Delta^+(h_i, s_i^b)$ is equal to $\Delta^+(H, b)$. This gives us a way to define the segments of H:

Lemma 7. *The length of each segment of H is equal to the sum of the lengths of the segments in the functions h_i with the same slope.*

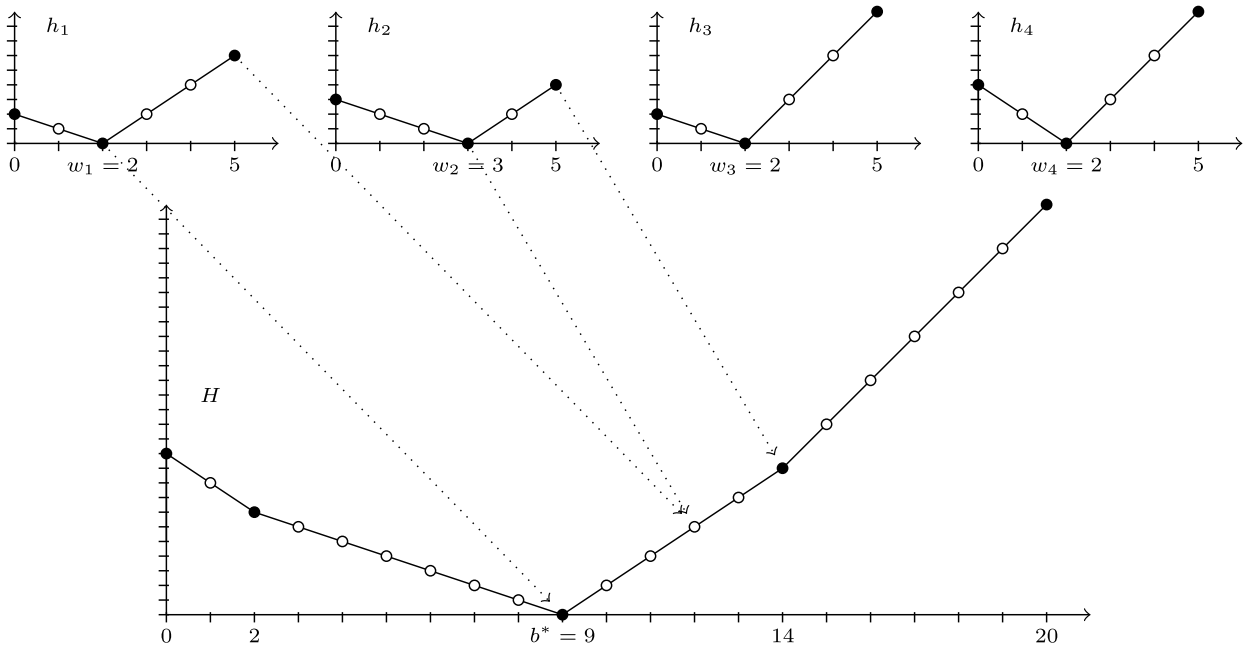

Fig. 3. The h_i and H functions of Example 8.

Proof. In the proof of Lemma 6, we saw that $\Delta^+(H, b)$ is equal to a minimal $\Delta^+(h_k, s_k^b)$. If one wants to increase b by more than one, then the increase per unit stays constant as long as there is at least one variable with slope equal to $\Delta^+(H, b)$. This defines a segment of slope $\Delta^+(H, b)$, whose length is equal to the sum of the lengths of the segments of all functions h_i with the same slope. □

We can use Lemmas 5 and 7 to construct H efficiently, as shown in the following example. Section 6.1 presents several ways to implement this construction in practice.

Example 8. Given the domain $[0, 5]$ for all x_i, the function H for the constraints of Example 3 can be constructed as follows (see also Fig. 3). Each h_i has two segments joining at w_i: the first spans $[0, w_i]$ and has slope $-r_i$, while the second spans $[w_i, 5]$ and has slope q_i. Starting from $b^* = 9$, we can define the segment of H for which b^* is the left breakpoint using the second segment of each h_i with minimal q_i. There are two of them (namely for $i = 1$ and $i = 2$) with respective lengths 3 and 2, both with slope 2. This defines a segment of length 5 and slope 2 spanning the interval $[9, 14]$. The next segment has slope 3 and is constructed from the second segments of h_i for $i = 3$ and $i = 4$. With a length of 6, it spans the interval $[14, 20]$. The same reasoning for values smaller than 9 leads to two more segments: one spans $[0, 2]$ with a slope of -2, and the other one spans $[2, 9]$ with a slope of -1. Fig. 3 shows how H is formed of the segments of the h_i. □

As can be seen in the previous example, we do not compute an analytical definition of H but only its minimum and its segments. As will be made clear in Section 6, we are never interested in the value of $H(b)$ for an arbitrary value b but only for b^* and for incremental modifications of b that can be computed using the slopes of the segments.

4.4. Computing the feasibility bound and a support

We now show that problem (6) can be solved in a greedy way.

Theorem 8. *Problem (6) can be solved greedily if each h_i is discretely convex.*

Proof. If each function h_i is discretely convex, then the function H is also discretely convex (by Theorem 4) and can be constructed from the h_i (by Lemmas 5 and 7). Finding the minimum of a discretely convex function under some bound constraints can be done greedily, as a local minimum of a discretely convex function is also a global minimum (see, e.g., Theorem 2.2 in [12]). □

Given the function H, problem (6) can be solved by first finding a value b minimising H (i.e., b^*), and then greedily increasing or decreasing b if b^* is not in $[\underline{g}, \overline{g}]$. In addition, computing a support $\mathbf{s}^b$ is useful for the filtering (to be discussed in Section 5).

Algorithm 1 Greedy algorithm to compute a support.

```
1:  function GetSupportLowerBound
2:      for all i ∈ [1, n] do                                   } initial bound
3:          s_i := argmin_{v ∈ g_i(D_{x_i})} h_i(v)
4:      b := Σ_{i∈[1,n]} s_i
5:      if b < g̲ then
6:          while Δ⁺(H, b) < +∞ and bp⁺(H, b) < g̲ do            } sharp bound
7:              b := bp⁺(H, b)
8:          if Δ⁺(H, b) = +∞ ∧ b < g̲ then return null
9:          Δ^max := Δ⁺(H, b)
10:         slack := g̲ − b
11:         for all i ∈ [1, n] do                               } modifying s
12:             while Δ⁺(h_i, s_i) < Δ^max do
13:                 s_i := bp⁺(h_i, s_i)
14:             if Δ⁺(h_i, s_i) = Δ^max and slack > 0 then
15:                 s' := min(bp⁺(h_i, s_i), s_i + slack)
16:                 slack := slack − s' + s_i
17:                 s_i := s'
18:     else if b > ḡ then
19:         while Δ⁻(H, b) < +∞ and bp⁻(H, b) > ḡ do            } sharp bound
20:             b := bp⁻(H, b)
21:         if Δ⁻(H, b) = +∞ ∧ b > ḡ then return null
22:         Δ^max := Δ⁻(H, b)
23:         slack := ḡ − b
24:         for all i ∈ [1, n] do                               } modifying s
25:             while Δ⁻(h_i, s_i) < Δ^max do
26:                 s_i := bp⁻(h_i, s_i)
27:             if Δ⁻(h_i, s_i) = Δ^max and slack < 0 then
28:                 s' := min(bp⁻(h_i, s_i), s_i + slack)
29:                 slack := slack − s' + s_i
30:                 s_i := s'
31:     return s
```

Thanks to Lemma 6, this can be achieved by Algorithm 1.[3] From now on, we write **s** to refer to $\mathbf{s}^b$. An assignment **s** that minimises the value of H without considering the bounds for b is initially constructed (lines 2–4). If b is in $[\underline{g}, \overline{g}]$, then the initial assignment is the final one. Otherwise the assignment is iteratively modified in order to satisfy the bounds of b. We assume $b < \underline{g}$ happens in line 5. Then some s_i must be increased until b is equal to $\underline{g}$. This is done in two steps. In lines 6–10, the segment of H where $\underline{g}$ lies is found. Its slope is stored in $\Delta^{\max}$, and the distance between $\mathrm{bp}^-(H, \underline{g})$ and $\underline{g}$ is stored in *slack*. Those two values allow us to modify each s_i separately (lines 11–17). For each i, first s_i is moved from breakpoint to breakpoint of h_i while the slope of the segment is smaller than $\Delta^{\max}$. Next, if the slope of the segment on the right of s_i is equal to $\Delta^{\max}$, then s_i is moved further on this segment, without exceeding the remaining slack (line 15). Lines 18–30 show the symmetrical case when $b > \overline{g}$: the left deltas and left breakpoints are used, and *slack* takes on negative values.

The algorithm returns the support **s** (line 31), or "null" if the constraint is unsatisfiable (lines 8 and 21), which triggers propagator failure and happens if there exists no value in the domains of the h_i such that $b \in [\underline{g}, \overline{g}]$.

We now prove that Algorithm 1 is correct:

Theorem 9. *If the h_i are discretely convex and H is defined as in equation (5), then Algorithm 1 returns an optimal solution to problem (4), and hence to problem (6), if one exists, and "null" otherwise.*

Proof. If the h_i are discretely convex, then H is also discretely convex (by Theorem 4) and lines 2–4 store its minimum in variable b (using Lemma 5). If $\underline{g} \le b \le \overline{g}$, then b is feasible and we are done. Otherwise, assume $b < \underline{g}$ (the other case is symmetrical), then by Theorem 2.2 in [12], the optimal solution is at $b = \underline{g}$. Now, we need to find $\mathbf{s}^{\underline{g}}$. We do this in two steps. Lines 6–7 locate the segment of H where $\underline{g}$ lies. If such a segment does not exist, then $\underline{g}$ is not in the domain of H

[3] Line 16 is corrected here with respect to our [1] where a sign error was present. Our implementation was correct.

Table 3
Values at some steps of Algorithm 1.

Step	b	*slack*	$\mathbf{s}$
after initial bound	9	–	$\langle 2,3,2,2\rangle$
after sharp bound	10	1	$\langle 2,3,2,2\rangle$
after modifying s_1	10	0	$\langle 3,3,2,2\rangle$

and "null" is returned in line 8. Otherwise, lines 9 and 10 compute the slope $\Delta^{\max}$ of that segment, and the distance *slack* between $\mathrm{bp}^-(H,\underline{g})$ and $\underline{g}$. Lines 11–17 modify the value of each w_i to reach $\underline{g}$: lines 12–13 position each s_i at the right breakpoint of the correct segment as defined by Lemma 6, and lines 14–17 ensure that $\sum_{i\in[1,n]} s_i$ is equal to $\underline{g}$. □

Example 9. We now show an execution of Algorithm 1 on the problem of our running example. Important values at some steps of Algorithm 1 are given in Table 3. We have already shown in Example 7 that $b^*=9$, so we have $b=9$ at line 4. As $b<\underline{g}=10$ on line 5, the algorithm enters the conditional branch starting at line 6 with $b=9$ and $\mathbf{s}=\langle 2,3,2,2\rangle$. We have that $\Delta^+(H,9)=2$ and $\mathrm{bp}^+(H,9)=14$. As $14\not< \underline{g}=10$, line 7 is never executed. Lines 9 and 10 set $\Delta^{\max}=2$ and $slack=1$. The loop of lines 11–17 is executed for each $i\in[1,4]$. Here, line 13 is never executed. For $i=1$, the condition of line 14 is true and lines 15–17 are executed: they set $s'=\min\{5,2+1\}=3$, $slack=1-3+2=0$, and $s^i=3$. As the slack is now equal to zero, no other value will be modified for $i\in[2,4]$. The final support is $\mathbf{s}=\langle 3,3,2,2\rangle$.

Note that we iterate in line 11 over the indices in increasing order. If we iterated in decreasing order, then we would obtain $\mathbf{s}=\langle 2,4,2,2\rangle$. The two assignments are both correct supports for $b=10$ and $H(b)=2$. The conjunction of constraints in this example is feasible as the optimal value, namely 2, is at most the upper bound $\overline{f}=5$. □

5. Domain filtering

To filter the domain of a variable, we extend the reasoning presented in Section 4.1. Indeed, variable x_j can take a value $u\in \mathrm{D}_{x_j}$ if and only if the cost of an optimal solution to the following problem is smaller than or equal to $\overline{f}$:

$$\begin{aligned} \text{minimise} \quad & f_j(u)+\sum_{i\neq j\in[1,n]} f_i(x_i) \\ \text{such that} \quad & \underline{g}\le g_j(u)+\sum_{i\neq j\in[1,n]} g_i(x_i)\le \overline{g} \\ & x_i\in \mathrm{D}_{x_i},\ \ \forall i\neq j\in[1,n] \end{aligned} \tag{8}$$

Problem (8) resembles problem (3) but x_j is fixed to u. Hence we can use the same reformulation as in Section 4.1. We introduce the following new function:

$$H_j(b)=\min\left\{\sum_{i\neq j\in[1,n]} h_i(y_i)\ \middle|\ \sum_{i\neq j\in[1,n]} y_i=b\wedge \forall i\neq j\in[1,n]: y_i\in g_i(\mathrm{D}_{x_i})\right\}$$

That is, $H_j(b)$ is similar to $H(b)$ in (5) but it only uses the functions h_i for i different from j. The optimal cost of problem (8) is the optimal cost of the following new problem:

$$\begin{aligned} \text{minimise} \quad & f_j(u)+H_j(z) \\ \text{such that} \quad & \underline{g}\le g_j(u)+z\le \overline{g} \end{aligned} \tag{9}$$

where value u is given and z is the only variable. The result of the following lemma can be used to compute H_j.

Lemma 10. *The function H_j is discretely convex if all h_i are discretely convex. The value b_j^* that minimises H_j is equal to the value b^* that minimises H minus the value v^* that minimises h_j. The length of each segment of H_j is equal to the length of the linear segment of H of the same slope minus the length of the linear segment of h_j of the same slope, if any.*

Proof. As H and H_j are defined identically except for the set of indices they consider, one can apply all results concerning H to H_j. So the first statement is a consequence of Theorem 4, replacing H by H_j. In the same way, applying Lemma 5 to H_j, we get $s_i^{b_j^*}=s_i^{b^*}$ for all $i\neq j$. Hence $b_j^*=b^*-s_i^{b^*}$, which proves the second statement. Applying Lemma 7 to H_j gives that the length of each segment of H_j is equal to the sum of the lengths of the segments of the same slope of the h_i with $i\neq j$. Hence the only difference between the length of a segment of H_j and the length of the segment of H of the same slope is the length of the segment of h_j with the same slope, if it exists. □

Algorithm 2 Main filtering algorithm.

```
function FilterTwoSums(f, g, h, f̄, g̲, ḡ)
    Construct H as discussed in Section 6.1
    s := GetSupportLowerBound
    for all j ∈ [1, n] do
        ForwardFilter(j)
        BackwardFilter(j)
```

Example 10. Given the functions of Example 3, H_1 is characterised as follows. Its minimum is at $9-2=7$ and it has 4 segments spanning respectively the interval $[0,2]$ with slope -2, the interval $[2,7]$ with slope -1, the interval $[7,9]$ with slope 2, and the interval $[9,15]$ with slope 3. Fig. 4 on page 183 illustrates the function and its use in the forthcoming Example 11. □

Our filtering algorithm is given in Algorithm 2. It iterates over all variables and, for each variable y_j, filters in turn for the values larger than s_j and for the values smaller than s_j. To avoid cluttering the algorithm descriptions with numerous parameters, we consider $\mathbf{f}, \mathbf{g}, \mathbf{h}, \overline{f}, \underline{g}, \overline{g}, H$, and $\mathbf{s}$ to be globally available. We show hereafter two ways to use H_j to implement ForwardFilter and BackwardFilter. The first way is applicable in general, provided H_j is discretely convex. The second way makes use of an additional property that f_j and g_j might have.

5.1. Filtering in the general case

As several values u of x_j can have the same image v through g_j, the set of values in D_{x_j} that are consistent with constraints (1) and (2) can be partitioned as:

$$\bigcup_{v \in g_j(\mathrm{D}_{x_j})} \left\{ u \;\middle|\; g_j(u) = v \wedge f_j(u) \leq \overline{f} - \min_{\underline{g} \leq z+v \leq \overline{g}} H_j(z) \right\}$$

That is, for each v, we have the set of values u in $g_j^{-1}(v)$ such that the optimal cost of problem (9) is at most $\overline{f}$, hence the set of values that are feasible. The domain of x_j can be made domain consistent by filtering the following unary constraint for each value $v \in g_j(\mathrm{D}_{x_j})$:

$$g_j(x_j) = v \Rightarrow f_j(x_j) \leq \overline{f} - \min_{\underline{g} \leq z+v \leq \overline{g}} H_j(z) \tag{10}$$

The function H_j being discretely convex, one can compute $\min_{\underline{g} \leq z+v \leq \overline{g}} H_j(z)$, which is independent of any particular u, incrementally from a value v to $v+1$. In addition, if v is equal to s_j, which is the value of y_j in the support $\mathbf{s}$ computed in Section 4.4, then

$$H_j\left(\sum_{i \neq j \in [1,n]} s_i\right) + h_j(s_j) = H\left(\sum_{i \in [1,n]} s_i\right) \tag{11}$$

This leads to Algorithm 3,[4] which is used to filter the domain of x_j for values larger than s_j. This algorithm traverses the functions h_j and H_j. The only complication is that in some cases (captured by the Boolean variable dec_b defined in lines 6 and 12) reaching an optimal solution to $\min_{\underline{g} \leq z+v \leq \overline{g}} H_j(z)$ involves decrementing b, which is the current value of z (line 10). Domain filtering according to constraint (10) takes place in lines 5 and 11. We assume that the Filter procedure to filter the domain of a variable for a unary constraint is provided by the user. The semantics of Filter$(\phi(x))$ for some unary predicate ϕ is that it removes from the domain of x all values u such that $\phi(u)$ does not hold. The algorithm ends when the optimal cost to problem (9) for $v+1$ is larger than $\overline{f}$ (line 8), at which point values of x_j for which $g_j(x_j) > v$ are filtered (line 14). The description of the procedure ComputeHj on line 2 will be given in Section 6.

Algorithm 4 presents the complementary algorithm for values smaller than s_j. In that case, v is iteratively decreased while b is increased. Hence the Boolean dec_b is replaced by inc_b defined as $b + v \leq \underline{g} \vee \Delta^+(H_j, b) < 0$. Note that line 5 is redundant with line 5 in Algorithm 3 but has been left for symmetry.

Theorem 11. *If each h_i is discretely convex, H is defined as in equation (5), $\mathbf{s}$ is the result of Algorithm 1, and each* Filter *call achieves domain consistency on the unary constraint given as argument, then Algorithms 3 and 4 achieve domain consistency on x_j.*

Proof. First, we show that the value b used in each Filter call of lines 5 and 11 is

[4] Line 8 is a clarification of our previously published version [1].

Algorithm 3 Filtering algorithm for values larger than s_j (general case).

1: **function** FORWARDFILTER(j)
2: $H_j :=$ COMPUTEHJ(H, h_j)
3: $b := \sum_{i\in[1,n]} s_i - s_j$
4: $v := s_j$
5: FILTER($g_j(x_j) = v \Rightarrow f_j(x_j) \leq \overline{f} - H_j(b)$)
6: $dec_b := b + v \geq \overline{g} \vee \Delta^-(H_j, b) < 0$
7: $valH_j :=$ if dec_b then $H_j(b-1)$ else $H_j(b)$
8: **while** $valH_j + h_j(v+1) \leq \overline{f}$ **do**
9: $v := v + 1$
10: **if** dec_b **then** $b := b - 1$
11: FILTER($g_j(x_j) = v \Rightarrow f_j(x_j) \leq \overline{f} - H_j(b)$)
12: $dec_b := b + v \geq \overline{g} \vee \Delta^-(H_j, b) < 0$
13: $valH_j :=$ if dec_b then $H_j(b-1)$ else $H_j(b)$
14: FILTER($g_j(x_j) \leq v$)

Algorithm 4 Filtering algorithm for values smaller than s_j (general case).

1: **function** BACKWARDFILTER(j)
2: $H_j :=$ COMPUTEHJ(H, h_j)
3: $b := \sum_{i\in[1,n]} s_i - s_j$
4: $v := s_j$
5: FILTER($g_j(x_j) = v \Rightarrow f_j(x_j) \leq \overline{f} - H_j(b)$)
6: $inc_b := b + v \leq \underline{g} \vee \Delta^+(H_j, b) < 0$
7: $valH_j :=$ if inc_b then $H_j(b+1)$ else $H_j(b)$
8: **while** $valH_j + h_j(v-1) \leq \overline{f}$ **do**
9: $v := v - 1$
10: **if** inc_b **then** $b := b + 1$
11: FILTER($g_j(x_j) = v \Rightarrow f_j(x_j) \leq \overline{f} - H_j(b)$)
12: $inc_b := b + v \leq \underline{g} \vee \Delta^+(H_j, b) < 0$
13: $valH_j :=$ if inc_b then $H_j(b+1)$ else $H_j(b)$
14: FILTER($g_j(x_j) \geq v$)

Table 4
Values at some steps of Algorithms 3 and 4, as per Example 11.

Step	b	v	dec_b/inc_b	$valH_j$	$h_j(v+1)$
Forward, after the initialisation	8	2	true	0	2
Forward, after one iteration	7	3	true	1	4
Forward, after two iterations	6	4	true	2	6
Backward, after the initialisation	8	2	true	4	1
Backward, after one iteration	7	3	true	7	2

$$\operatorname*{argmin}_{\underline{g}\leq b+v\leq\overline{g}} H_j(b) \tag{12}$$

This is verified for the call in line 5 by equation (11) and the computed values of b and v. From now on, we only consider the case of the values v larger than w_j, i.e., Algorithm 3, the other case being symmetrical. Each iteration of the loop of lines 8–13 increments v by 1. If $b+v=\overline{g}$, then b must be decreased by 1 in order to satisfy the condition in expression (12) when v is incremented. If $\Delta^-(H_j, b) < 0$, then $H_j(b-1) < H_j(b)$. The value $b-1$ is not feasible for v (as b is optimal, meaning that $b+v=\underline{g}$) but is feasible for $v+1$, so b must be decremented in that case. In all other cases, b stays constant. Hence, in each case, the value of b at line 11 is equal to the value of expression (12).

Second, we show that the value v in line 14 is the largest feasible value in the domain of y_j. By the argument above, b is the optimal value at each iteration of the loop. Hence the test of line 8 evaluates to false if and only if $v+1$ is not feasible. As H_j is discretely convex, there is no feasible value larger than v. □

We note that domain consistency is usually easy to achieve for any unary constraint appearing in a FILTER call.

In Section 7.1, we will characterise the consistency level achieved by Algorithms 3 and 4 when the h_i are not discretely convex.

Example 11. The execution of Algorithm 3 on variable x_1 in the problem of Example 3 is as follows. Table 4 gives the values of the variables appearing in the algorithm at different moments and Fig. 4 shows how the functions h_j and H_j are traversed. We start from the support $\mathbf{s} = \langle 2, 4, 2, 2\rangle$ and have $\overline{f} = 5$. The result of line 2, namely H_j, was given in Example 10. Lines 3 and 4 set $b = 8$ and $v = 2$. The FILTER calls at lines 5 and 11 amount, after instantiating and simplifying the formula, to "if $\max\{2-v, 2v-4\} + H_j(b) > 5$, then remove v from D_{x_1}". In line 5, we have $H_j(b) = 2$, hence $v = 2$

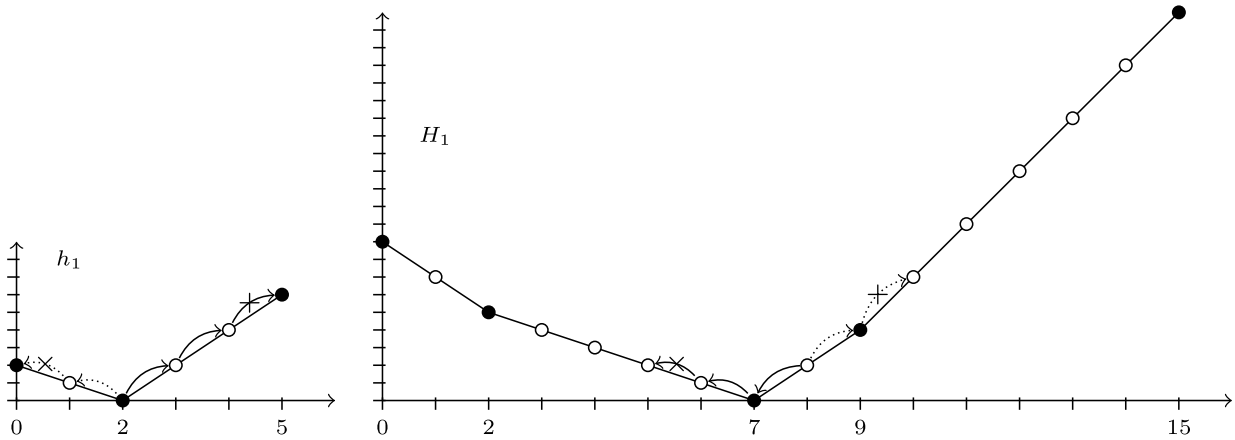

Fig. 4. The h_1 and H_1 functions used in Examples 10 and 11. The solid arrows correspond to the successive iterations of Algorithm 3. The loop ends after two iterations as $H_1(5)+h_1(5)=2+6=8>5$. The dotted arrows correspond to the successive iterations of Algorithm 4. The loop ends after one iteration as $H_1(10)+h_1(0)=7+2=9>5$.

stays in the domain. Line 6 sets dec_b to true. As $\underline{g}=\overline{g}$ in this example, dec_b is always true and b will be decremented at each iteration. The tested value in line 8 is equal to $0+2=2$, which is smaller than 5, hence the loop is entered. Now, we have $v=3$, $b=7$, and $H_j(b)=0$, and nothing is filtered by line 11. To start the next iteration, we have that the tested value in line 8 is $1+4=5$ and the body of the loop is executed again: $v=4$, $b=6$, $H_j(b)=1$, and nothing is filtered. Finally, the tested value in line 8 becomes $2+6=8$, which is larger than 5, and the loop ends. The execution of line 14 enforces $x_1 \le v=4$, which removes 5 from D_{x_1}. The domain of x_1 after the execution of Algorithm 3 is thus $[0,4]$.

The execution of Algorithm 4 is similar and illustrated with the dotted arrows in Fig. 4. The loop ends when $v=1$ and the execution of line 14 enforces $x_1 \ge v=1$, which removes 0 from D_{x_1}. The domain of x_1 after the execution of Algorithm 4 is thus $[1,4]$. □

5.2. Filtering in a special case

We now present a special case that allows us to avoid useless computation, namely a form of monotonicity. Let us define $k_j(v)=\max f_j(g_j^{-1}(v))$, that is $k_j(v)$ is the largest value $f_j(u)$ for u such that $g_j(u)=v$. The function k_j is similar to h_j but the 'max' operator replaces the 'min' one.

If $h_j(v) \ge k_j(v-1)$ for any value v larger than $v^* = \mathrm{argmin}_{u \in g_j(\mathrm{D}_{x_j})} h_j(u)$ and $h_j(v) \ge k_j(v+1)$ for any v smaller than v^*, then there exists a value $v^{\max}$ such that for all values $v \in g_j(\mathrm{D}_{x_j})$ smaller than $v^{\max}$ (but larger than or equal to s_j) we have that all values $u \in g_j^{-1}(v)$ are feasible, and for all v larger than $v^{\max}$, there is no feasible u. We then need not consider all values but only find $v^{\max}$ and filter according to the two constraints $g_j(x_j) \le v^{\max}$ and $g_j(x_j)=v^{\max} \Rightarrow f_j(x_j) \le \overline{f} - \min_{\underline{g} \le z+v^{\max} \le \overline{g}} H_j(z)$. A similar argument holds for a similarly defined $v^{\min}$.

Finding $v^{\max}$ amounts to computing the largest value v such that $h_j(v)+\min_{\underline{g} \le z+v \le \overline{g}} H_j(z) \le \overline{f}$. As h_j and H_j are both convex, this problem can be solved by incrementally increasing v until the bound is reached. Algorithm 5 presents the steps to find $v^{\max}$.[5] This algorithm is very similar to Algorithm 3, but it does not need to iterate over all the values v, but only over the ones that are at a breakpoint of h_j or H_j. The increment is stored in ℓ (lines 6, 12, and 14). For conciseness, we use **if-then-else** expressions inside the expressions giving the value of ℓ in lines 6, 12, and 14. The correctness of Algorithm 5 stems from the correctness of Algorithm 3 and the fact that ℓ is the largest increment such that the values of $\Delta^+(h_j,v)$, $\Delta^-(H_j,b)$, and dec_b are constant.

An example of the special case is when g_j is the identity function. Then g_j is injective. Hence $h_j=k_j$ and, by convexity, h_j is non-decreasing to the right of v^* and non-increasing to the left of v^*.

Example 12. The filtering of Example 11 can be rerun with Algorithm 5 since the functions g_j are the identity function. Table 5 gives the values of the variables appearing in the algorithm at different moments. Initially, we have $v=2$, $b=8$, and $\ell=8-7=1$, as H_j has a breakpoint at $b=7$. After one iteration, we have $v=3$, $b=7$, and $\ell=5-3=2$, as h_j has a breakpoint at 5. At this point, the tested value in line 8 is $2+6=8$, and the loop ends. Lines 14 and 15 set $\ell=(5-0-2)/(2+1)=1$ and $v=4$. As in Example 11, line 16 removes all values larger than 4 (i.e., 5) from D_{x_1}. Line 17 does not remove anything here. Similarly, the execution of Algorithm 6 stops after one iteration and removes value 0 from the domain of x_1. □

[5] Lines 6 and 12 have been corrected and line 8 is a clarification of our previously published version [1]. Our implementation was correct.

Algorithm 5 Filtering algorithm for values larger than s_j (special case).

```
function ForwardFilter(j)
    H_j := computeHj(H, h_j)
    b := Σ_{i∈[1,n]} s_i − s_j
    v := s_j
    dec_b := b + v ≥ ḡ ∨ Δ⁻(H_j, b) < 0
    ℓ := min(bp⁺(h_j, v) − v, if dec_b then b − bp⁻(H_j, b) else ḡ − b − v)
    valH_j := if dec_b then H_j(b − ℓ) else H_j(b)
    while valH_j + h_j(v + ℓ) ≤ f̄ do
        v := v + ℓ
        if dec_b then b := b − ℓ
        dec_b := b + v ≥ ḡ ∨ Δ⁻(H_j, b) < 0
        ℓ := min(bp⁺(h_j, v) − v, if dec_b then b − bp⁻(H_j, b) else ḡ − b − v)
        valH_j := if dec_b then H_j(b − ℓ) else H_j(b)
    ℓ := (f̄ − H_j(b) − h_j(v))/(Δ⁺(h_j, v) + (if dec_b then Δ⁻(H_j, b) else 0))
    v := v + ℓ
    Filter(g_j(x_j) ≤ v)
    Filter(g_j(x_j) = v ⇒ f_j(x_j) ≤ f̄ − H_j(b))
```

Table 5
Values at some steps of Algorithm 5, as per Example 12.

Step	b	v	dec_b	ℓ	$valH_j$	$h_j(v-1)$
after the initialisation	8	2	true	1	0	2
after one iteration	7	3	true	2	2	6

Algorithm 6 Filtering algorithm for values smaller than s_j (special case).

```
function BackwardFilter(j)
    H_j := computeHj(H, h_j)
    b := Σ_{i∈[1,n]} s_i − s_j
    v := s_j
    inc_b := b + v ≤ g̲ ∨ Δ⁺(H_j, b) < 0
    ℓ := min(v − bp⁻(h_j, v), if inc_b then bp⁺(H_j, b) − b else b + v − g̲)
    valH_j := if inc_b then H_j(b + ℓ) else H_j(b)
    while valH_j + h_j(v − ℓ) ≤ f̄ do
        v := v − ℓ
        if inc_b then b := b + ℓ
        inc_b := b + v ≤ g̲ ∨ Δ⁺(H_j, b) < 0
        ℓ := min(v − bp⁻(h_j, v), if inc_b then bp⁺(H_j, b) − b else b + v − g̲)
        valH_j := if inc_b then H_j(b + ℓ) else H_j(b)
    ℓ := (f̄ − H_j(b) − h_j(v))/(Δ⁻(h_j, v) + (if inc_b then Δ⁺(H_j, b) else 0))
    v := v − ℓ
    Filter(g_j(x_j) ≥ v)
    Filter(g_j(x_j) = v ⇒ f_j(x_j) ≤ f̄ − H_j(b))
```

Algorithm 6 presents the complementary algorithm to find $v^{\min}$. Again, it is very similar to Algorithm 4 but, like Algorithm 5, it decreases the value of v in each step by ℓ, which is potentially larger than 1.

6. A parametric propagator and its complexity

Our propagator is generic in the sense that it works correctly for any functions f_i and g_i that respect the condition of Theorem 8. However, we call it a *parametric* propagator, because rather than resorting to a fully generic implementation, we use hook functions and procedures that need to be provided. This often allows us to get a lower time complexity. The parameters for each instantiation are shown in Table 6: they are used in Algorithms 1 to 6. We now study the time and space complexity of our propagator, after describing some implementation strategies.

6.1. Constructing H

We represent the H function as two linked lists of segments, plus two integers for the value b^* minimising $H(b^*)$ and for $H(b^*)$ itself. For each segment, its slope and length are stored. One linked list chains all the segments to the right of b^* by increasing order of the slope value and is terminated by a dummy segment with slope $+\infty$. The other linked list chains the segments to the left of b^* by decreasing order of the slope value and is terminated by a dummy segment with slope $-\infty$.

Constructing the linked lists of H, i.e., line 2 in Algorithm 2, can be implemented in various ways.

Table 6
Parameters to instantiate.

Functions	Procedures
$\text{argmin}_{v \in g_i(\mathrm{D}_{x_i})} h_i(v)$	$\textsc{Filter}(g_i(x_i) \leq v)$
$\min_{v \in g_i(\mathrm{D}_{x_i})} h_i(v)$	$\textsc{Filter}(g_i(x_i) \geq v)$
$\Delta^+(h_i, v)$	$\textsc{Filter}(g_i(x_i) = v \Rightarrow f_i(x_i) \leq u)$
$\Delta^-(h_i, v)$	
$\text{bp}^+(h_i, v)$	
$\text{bp}^-(h_i, v)$	

Algorithm 7 Maintaining $H(b)$, s, and l when incrementing b by k.

In/Out: b ▷ The identifier b appearing in Algorithm 1
In/Out: H_b ▷ The current value of $H(b)$
In/Out: s ▷ The segment of H on which b and $b+1$ lie
In/Out: ℓ ▷ The distance between b and the right breakpoint of s
Require: $k \leq \ell$
1: **procedure** $\textsc{Increment}_{H,b}(k)$
2: $b := b + k$
3: $H_b := H_b + k \cdot \Delta(s)$ ▷ Δ returns the slope of the segment
4: $\ell := \ell - k$
5: **if** $\ell = 0$ **then**
6: $s :=$ SUCC(s) ▷ SUCC returns the next segment in the linked list
7: $\ell :=$ LENGTH(s) ▷ LENGTH returns the length of the segment

Algorithm 8 Maintaining $h_j(v)$ when incrementing v by k.

In/Out: v ▷ The identifier v appearing in Algorithms 3 to 6
In/Out: h_j_v ▷ The current value of $h_j(v)$
Require: $k \leq \text{bp}^+(h_j, v) - v$
1: **procedure** $\textsc{Increment}_{h_j,v}(k)$
2: $v := v + k$
3: $h_j_v := h_j_v + k \cdot \Delta^+(h_j, v)$ ▷ $\Delta^+(h_j, v)$ is a parameter to instantiate

A first way is to traverse each function h_i in turn and to build H incrementally by traversing the linked lists in parallel. This takes $\mathcal{O}(n \cdot (s(h) \cdot p + s(H)))$ time, where $s(h)$ is the maximum number of segments among the h_i functions, $s(H)$ is the number of segments of H, and p is the highest complexity of the parametric functions in Table 6.

A second way is to collect all the segments from all the functions in a list, to sort this list, and to construct H by traversing the list. This takes $\mathcal{O}(n \cdot s(h) \cdot (p + \log(n \cdot s(h))))$ time and is asymptotically better than the first way when $s(H) > s(h) \cdot \log(n \cdot s(h))$.

Although it might be interesting to construct H lazily because some parts of H might never be used, preliminary experiments have shown that the construction of H takes only a very small portion of the running time so that we did not explore this direction further.

6.2. Computing $H(b)$, $h_j(v)$, and $H_j(b)$

The value of $H(b)$ is never queried for arbitrary values of b, but only for a value b^* minimising $H(b^*)$ and for incrementally modified values of b, so that $H(b)$ can also be computed incrementally.

The computation of $H(b)$, $\Delta^+(H, b)$, $\Delta^-(H, b)$, $\text{bp}^+(H, b)$, and $\text{bp}^-(H, b)$ can be performed as follows in constant time for all values of b used in the algorithms. Note first that in any algorithm the value of b is either only increased or only decreased, starting from b^*. We only discuss here the case of increasing b, the decreasing case being symmetrical. At any point in the algorithms, instead of maintaining only the value b, we also maintain a pointer to the segment s of H in which b and $b+1$ lie (there is always a unique such segment), the distance ℓ between b and the right breakpoint of s, and the value of $H(b)$.

Algorithm 7 shows how those quantities can be maintained upon incrementing b. If b is incremented by k with $k \leq \ell$ (which is ensured in all algorithms), then $H(b)$ is incremented by $k \cdot \Delta(s)$ and ℓ is decreased by k. If ℓ becomes equal to 0, then s is replaced by its successor in the linked list and ℓ is set to the length of the new segment. The expression $\Delta^+(H, b)$ appearing in Algorithm 1 is equal to $\Delta(s)$ and $\text{bp}^+(H, b)$ can be computed as $b + \ell$.

Similarly, the value of $h_j(v)$ is only queried for $s_i^{b^*}$ and incrementally modified values of v. This is reflected by the absence of $h_j(v)$ from the parameters in Table 6. As shown in Algorithm 8, the bookkeeping associated with h_j is simpler than the one of H: it suffices to maintain $h_j(v)$ using the parameter function $\Delta^+(h_j, v)$ whenever v is modified.

The approach for querying the value of $H_j(b)$ is the same as for $H(b)$: the value of b is initialised to a value for which $H_j(b)$ is known, and then either only increased or only decreased. Hence $H_j(b)$ and the other related quantities can be maintained incrementally as is done for H. Using this fact and Lemma 10, we actually do not need to compute H_j in line 2 of Algorithms 3 to 6. Instead, we only maintain the segment s of H_j on which b and $b+1$ lie (in case of increasing b). As

Algorithm 9 Maintaining $H_j(b)$ and related quantities when incrementing b by k.

```
In/Out: b                                  ▷ The identifier b appearing in Algorithms 3 to 6
In/Out: H_j_b                              ▷ The current value of H_j(b)
In/Out: d                                  ▷ The slope of the segment of H_j on which b and b + 1 lie
In/Out: ℓ                                  ▷ The distance between b and its right breakpoint in H_j
In/Out: v′                                 ▷ A value such that −Δ⁻(h_j, v) ≤ d ∧ Δ⁺(h_j, v) > d
In/Out: s′                                 ▷ The segment of H on which b + v′ and b + v′ + 1 lie
Require: k ≤ ℓ
 1: procedure INCREMENT_{H_j,b}(k)
 2:   b := b + k
 3:   H_j_b := H_j_b + k · d
 4:   ℓ := ℓ − k
 5:   while ℓ = 0 do
 6:     s′ := SUCC(s′)
 7:     d := Δ(s′)
 8:     ℓ := LENGTH(s′)
 9:     if d = Δ⁺(h_j, v′) then
10:       ℓ := ℓ − (bp⁺(h_j, v′) − v′)
11:       v′ := bp⁺(h_j, v′)
```

Table 7
Time complexity of the different versions of the propagator.

Case	Construct H	Time complexity
General	Traversing	$\mathcal{O}(n \cdot (s(h) \cdot p + s(H) + r(h) \cdot c))$
	Sorting	$\mathcal{O}(n \cdot (s(h) \cdot p + s(h) \cdot \log(n \cdot s(h)) + r(h) \cdot c))$
Special	Traversing	$\mathcal{O}(n \cdot (s(h) \cdot p + s(H) + c))$
	Sorting	$\mathcal{O}(n \cdot (s(h) \cdot p + s(h) \cdot \log(n \cdot s(h)) + s(H) + c))$

shown in Algorithm 9, to compute the length ℓ and slope d of s, we also maintain a pointer to the segment s' in H from which the segment of H_j is built and a value v' such that $-\Delta^-(h_j, v') \leq \Delta(s) < \Delta^+(h_j, v')$. When we need to access the next segment of H_j (because the value ℓ reached 0), we can construct it from the successor of s' in H and the segment on the right of v' in h_j. As a segment in H might be built from a single segment of only h_j, Algorithm 9 must loop while the remaining length ℓ is equal to zero.

6.3. Complexity analysis

Feasibility test Algorithm 1 computes a support in $\mathcal{O}(s(H) + n \cdot s(h))$ time, that is $\mathcal{O}(n \cdot s(h))$ time, as $s(H) \leq n \cdot s(h)$. This is dominated by the prior construction of H discussed in Section 6.1.

Filtering We implement Algorithms 3 and 4 to run in $\mathcal{O}(r(h) \cdot (p + c))$ time, where $r(h) = |g_j(\mathrm{D}_{x_j})|$, p is the highest complexity of the parametric functions in Table 6, and c is the highest complexity of the FILTER procedures in Table 6. The segments of H_j are computed on the fly from h_j and H, as explained in Section 6.2. The sum in line 3 of Algorithms 3 and 4 is actually provided by our representation of H, so it need not be recomputed each time. Algorithms 5 and 6 take $\mathcal{O}(s(h) \cdot p + s(H) + c)$ time.

The whole propagator The time complexity of our propagator, given in Algorithm 2, is obtained by multiplying the filtering complexity by n (the number of variables) and adding the complexity of computing H. Table 7 summarises this for the different versions of the propagator. Note that $s(h) \leq r(h) \leq |\mathrm{D}_x|$ and $s(H) \leq n \cdot s(h)$. In the worst case, one can assume that $p = c = \mathcal{O}(d)$, where d is the size of the largest domain, leading to the complexity announced in Section 1 for the general case and constructing H by traversal: $\mathcal{O}(n \cdot d^2 + n^2 \cdot d)$. However, we show in Table 1 on page 171 and in Section 8 that the time complexity for specific instantiations of the propagator is much lower.

The space complexity of our propagator is $\mathcal{O}(n + s(H))$, as we need to store a constant amount of information for each variable (namely the value s_i), as well as the whole function H (which amounts to a constant amount for each of its $s(H)$ segments). The functions h_i and H_j are not stored explicitly.

7. Instantiating the parametric propagator

We now present a number of relaxations and extensions of the problem as covered in the previous sections. At the same time, we discuss the consistency levels that can be achieved by our propagator. Those results are used in Section 8 when instantiating the parametric propagator for particular pairs of constraints.

7.1. Relaxations

The required discrete convexity of the h_i functions puts a strong restriction on the shape of the g_i. Recall that $g_i(\mathrm{D}_{x_i})$ must be an interval by the first condition in Definition 1. As the discrete convexity must be respected for *all* D_{x_i} that arise during the search, the only instantiations of g_i satisfying the first condition of Definition 1 are those whose image contains only two values, which must be one unit apart. We call these *characteristic functions*. In such a case, the second condition of Definition 1 is always respected and the f_i can be any (integer) functions. This is for instance the case of the constraint pairs Linear$_\leq$ and Among, and Linear$_\leq$ and Maximum shown in Table 2 on page 175.

If D_{x_i} is restricted to be an interval, then the class of g_i functions satisfying the first condition of Definition 1 is more general, namely all functions where

$$|g_i(u) - g_i(u+1)| \leq 1 \quad \forall u, u+1 \in \mathrm{D}_{x_i} \tag{13}$$

If there are holes in a domain D_{x_i}, then D_{x_i} can be relaxed to the smallest enclosing interval, but some propagation may be lost: this compromise is often acceptable for global constraints. In this case, we do not achieve domain consistency, but bounds($\mathbb{Z}$) consistency. Among others, the identity function respects equation (13). If some g_i is the identity function, then f_i must be discretely convex, because $h_i = f_i$.

In general, if h_i is not discretely convex, then one can replace it by a discretely convex function $h'_i \colon S \to T$ that *underapproximates* h_i, i.e., such that $g_i(\mathrm{D}_{x_i}) \subseteq S$ and $h'_i(v) \leq h_i(v)$ for all $v \in g_i(\mathrm{D}_{x_i})$. Using h'_i, the propagator remains correct but might miss propagation.

Even when h_i is discretely convex, it can be beneficial to replace it by an underapproximation in order to reduce the time complexity of the algorithms, at the sacrifice of a potentially weaker filtering. For example, a generally applicable relaxation is to replace all the segments of a function h_i on the right of some s_i^b by a single segment with slope $\Delta^+(h_i, s_i^b)$, and all segments on the left of s_i^b by a single segment with slope $-\Delta^-(h_i, s_i^b)$. This reduces the term $s(h)$ in the complexity analysis to the constant 2 but may lead to some missed filtering.

7.2. Extensions

If some h_i is a linear function, then $-h_i$ is also discretely convex. Hence, one can put a lower bound $\underline{f}$ on $\sum_{i\in[1,n]} f_i(x_i)$ and run the propagator twice, first with constraint (1) being $\sum_{i\in[1,n]} f_i(x_i) \leq \overline{f}$, and then with constraint (1) being $-\sum_{i\in[1,n]} f_i(x_i) \leq -\underline{f}$.

Our propagator can be extended to handle variables as the upper and lower bounds of the constraints. In such a case, the largest values in the domains of $\overline{f}$ and $\overline{g}$, as well as the smallest values in the domains of $\underline{f}$ and $\underline{g}$ are used in the propagator. In addition, the other bound of each variable can be constrained using the H function without changing the time complexity. Only bounds($\mathbb{Z}$) consistency can be achieved on those variables.

Our propagator can also be adapted to work with the f_i being functions from integers to *reals*. As long as the g_i are defined from integers to integers, the domains of the intermediate variables y_i and z stay subsets of the integers. However, care must be taken when implementing operations on reals using floating point numbers.

8. Example instantiations

We now show that many existing (pairs of) constraints fit our parametric constraint, optionally relaxed or extended as in Section 7. Table 2 on page 175 presents several instantiations of the f_i and g_i, together with the derived h_i. We discuss below some of these constraints and the time complexity of the parametric propagator in those cases, also summarised in Table 1 on page 171. The instantiated complexities are derived from the parametric complexities in Table 7 by replacing $s(h)$, $s(H)$, $r(h)$, p, and c by suitable values derived from the h_i.

If $g_i(u) = 0$ for all i, then the second constraint vanishes and we can use our propagator for a single Sum$_\leq$ constraint. In such a case, our parametric propagator achieves domain consistency in $\mathcal{O}(n \cdot (p + c))$ time. For the particular case of a linear inequality (Linear$_\leq$), our parametric propagator runs in $\mathcal{O}(n)$ time. Although this complexity matches the theoretical complexity of a dedicated propagator, our propagator is too general for this simple case and does not use any practical improvement such as the ones presented in [10].

Similarly, if $f_i(u) = 0$ for all i, then the first constraint vanishes and we can use our propagator for a single Sum$_=$ constraint, with $\underline{g} = \overline{g}$, achieving bounds($\mathbb{R}$) consistency in $\mathcal{O}(n \cdot d)$ time. Again, for Linear$_=$, we match the theoretical $\mathcal{O}(n)$ complexity of dedicated propagators but without any practical improvement such as the ones presented in [10].

The case $g_i(u) = u$ for all i covers many interesting constraints already presented in the literature. In particular, it covers the bounds($\mathbb{Z}$)-consistent propagators for the statistical constraints Deviation and Spread with a fixed rational average. Interestingly, it can be generalised to any L_p-norm, with $p > 0$, except $\mathrm{L}_{+\infty}$. One can also give a different penalty for deviations over and under the average. This may be very useful as in many practical situations it is less problematic to deviate from the average in one direction than in the other. The time complexity of our propagator is $\mathcal{O}(n)$ for Deviation, which matches the best published propagator [3]. For Spread and higher norms, the time complexity of our propagator is $\mathcal{O}(n \cdot d_\cup)$, with $d_\cup = \left|\cup_{i\in[1,n]} \mathrm{D}_{x_i}\right|$. This is incomparable to the $\mathcal{O}(n \cdot \log n)$ complexity of the best published propagator [2].

Table 8
Expressions for instantiating a propagator for Deviation. The conditions are not always mutually exclusive and are to be evaluated in top-down order.

Parameter	Instantiation
$\text{argmin}_{v \in g_i(\mathrm{D}_{x_i})} h_i(v)$	$\begin{cases} \lceil \mu \rceil & \text{if } \min \mathrm{D}_{x_i} \le \mu \le \max \mathrm{D}_{x_i} \wedge \lceil \mu \rceil - \mu < \mu - \lfloor \mu \rfloor \\ \lfloor \mu \rfloor & \text{if } \min \mathrm{D}_{x_i} \le \mu \le \max \mathrm{D}_{x_i} \wedge \lceil \mu \rceil - \mu \ge \mu - \lfloor \mu \rfloor \\ \min \mathrm{D}_{x_i} & \text{if } \mu < \min \mathrm{D}_{x_i} \\ \max \mathrm{D}_{x_i} & \text{if } \mu > \max \mathrm{D}_{x_i} \end{cases}$
$\Delta^+(h_i, v)$	$\begin{cases} +\infty & \text{if } v = \max \mathrm{D}_{x_i} \\ -n & \text{if } v < \lfloor \mu \rfloor \\ n \cdot (\lceil \mu \rceil + \lfloor \mu \rfloor) - 2 \cdot n \cdot \mu & \text{if } v = \lfloor \mu \rfloor \wedge \lfloor \mu \rfloor \ne \lceil \mu \rceil \\ n & \text{if } v \ge \lceil \mu \rceil \end{cases}$
$\mathrm{bp}^+(h_i, v)$	$\begin{cases} +\infty & \text{if } v = \max \mathrm{D}_{x_i} \\ \min\left(\max \mathrm{D}_{x_i},\ \lfloor \mu \rfloor\right) & \text{if } v < \lfloor \mu \rfloor \\ \lceil \mu \rceil & \text{if } v = \lfloor \mu \rfloor \wedge \lfloor \mu \rfloor \ne \lceil \mu \rceil \\ \max \mathrm{D}_{x_i} & \text{if } v \ge \lceil \mu \rceil \end{cases}$
$\textsc{Filter}(g_i(x_i) \le v)$	$\textsc{Filter}(x_i \le v)$
$\textsc{Filter}(g_i(x_i) = v \Rightarrow f_i(x_i) \le u)$	$\textsc{Filter}(\lvert n \cdot v - n \cdot \mu \rvert > u \Rightarrow x_i \ne v)$

Note that our propagator achieves bounds($\mathbb{Z}$) consistency, which has only been achieved recently and independently in the case of Spread [11].

As an example, we show in Table 8 the instantiations of the parameters for Deviation (symmetric parameters are omitted): note that each h_i has (up to) three segments, joining at the breakpoints $\lfloor \mu \rfloor$ and $\lceil \mu \rceil$.

If g_i is a characteristic function, then f_i can be any function. A characteristic function may be used to count, as in the case of the Count family of constraints (e.g., Among [14,15]). But characteristic functions can also be used to represent the Maximum constraint with a fixed maximum m. Indeed, the constraint $m = \max_{i \in [1,n]} x_i$ can be decomposed into $\forall i \in [1, n]: m \ge x_i \wedge \sum_{1 \in [1,n]} [x_i = m] \ge 1$. Table 2 gives a definition of the functions h_i for $\textsc{Linear}_{\le}$ and Exactly, in which case our propagator achieves domain consistency and runs in $\mathcal{O}(n \cdot (\log n + p + c))$ time, as does the dedicated propagator presented in [9].

Many other pairs of Sum constraints can be instantiated. Note that the functions f_i and g_i can differ for each i, i.e., one can mix in the same sum terms of different forms (e.g., some linear and some quadratic), as long as each corresponding function h_i is discretely convex.

Example 13. We now complete our running example. Applying Algorithm 2 achieves bounds($\mathbb{Z}$) consistency and yields the following reduced domains: $x_1 \in [1, 4]$, $x_2 \in [2, 5]$, $x_3 \in [1, 3]$, and $x_4 \in [1, 3]$. In total, 9 values were removed from the domains. In contrast, if the two Sum constraints were filtered separately, then the domains would be $x_1 \in [0, 4]$, $x_2 \in [0, 5]$, $x_3 \in [0, 3]$, and $x_4 \in [0, 3]$, for a total of only 5 removed values.

Generalising to any number n of workers, our parametric propagator takes $\mathcal{O}(n^2)$ time. This is obtained by instantiating the time complexity given in Table 7 for the special case with $p = c = \mathcal{O}(1)$ as all parametric functions and procedures can be implemented in constant time, $s(h) = 2$ as each h_i function has 2 segments, and $s(H) = 2 \cdot n$ as, in the worst case, the parameters **r** and **s** are all different. □

9. Experimental evaluation

To show that the parametricity of our propagator is not detrimental not only to asymptotic complexity (as seen in Section 8) but also to efficiency, we make an experiment to compare custom propagators with instantiations of our parametric propagator. We selected the Deviation [3] and Spread [11] constraints as their bounds($\mathbb{Z}$)-consistent propagators are freely available in the distribution of OscaR [16]. We implemented our propagator and its instantiations on top of OscaR. We performed the comparison on the 100 instances of the balanced academic curriculum problem (BACP) that were introduced in [17],[6] modelled as in the OscaR distribution (we only slightly modified the search heuristic in order to make it deterministic, so that the search trees are the same for comparison purposes).

For Deviation, we used the 44 instances that are solved to optimality in more than 1 second (to avoid measurement errors) but less than 12 hours (3 instances timed out). When using our parametric propagator, the time to solve an instance is on average only 7% longer than when using the custom propagator, with a standard deviation of 5%: see also Fig. 5, left. The numbers of nodes in the search tree and calls to the propagator are *exactly* the same for both propagators due to their common level of consistency and the deterministic search procedure.

For Spread, we used the 33 instances that are solved to optimality in more than 1 second but less than 12 hours (2 instances timed out). When using our parametric propagator, the time to solve an instance is on average 28% *shorter* than

[6] They are available at http://becool.info.ucl.ac.be/resources/bacp.

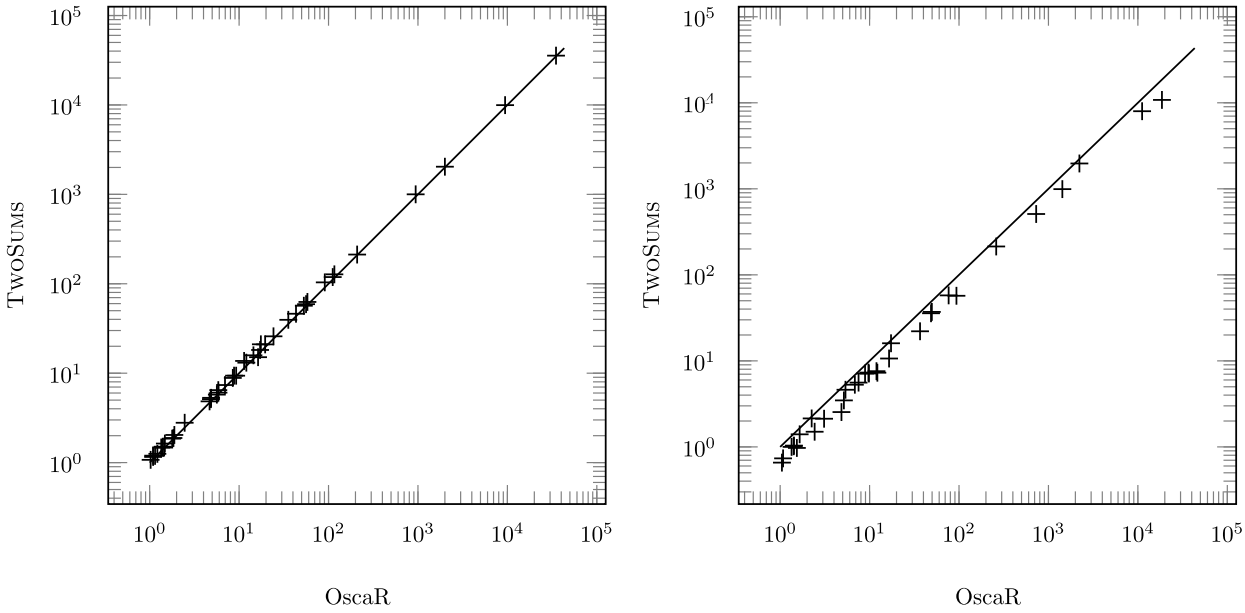

Fig. 5. Results for Deviation (left) and Spread (right) comparing the time in seconds for solving an instance using either the specialised propagator or our propagator. We only report instances that take more than 1 second but less than 12 hours.

when using the custom propagator, with a standard deviation of 10%: see also Fig. 5, right. This improvement is explained by a different algorithmic approach, which is in our favour when the domains of the variables are small, as is the case for the BACP instances. The numbers of nodes in the search tree and calls to the propagator are *exactly* the same for both propagators.

Our Java implementation and the raw data for the results reported here are available at http://www.it.uu.se/research/group/astra/software#convexpairs. A package for replication is at http://recomputation.org [18].

10. Related work

Our approach of first computing a feasibility bound and then incrementally adapting it is not new and has been used in the design of several propagators. Among others, this is the case for the cited propagators covered by our own propagator. However, the novelty of our work is that for the first time we abstract from the details of each constraint to focus on their common properties. This is close in spirit to what has been done with SeqBin [19,20] for another class of constraints.

When the g_i are characteristic functions, our conjunction of Sum constraints can also be represented using CostGCC [21]. However, this would require the explicit representation of all variable–value pairs and induce a higher time complexity than our propagator. On the other hand, CostGCC can handle more than one counting constraint in one propagator.

To the best of our knowledge, the notion of discrete convexity has not been used before for the design of propagators. However, several researchers have exploited other forms of convexity in constraint programming, namely row convexity [22] and tree convexity [23]. Those forms of convexity are unrelated to the one considered here and the focus of the cited works is on global properties of the constraint network rather than a propagator.

Finally, it should be noted that there are domain-consistent propagators for several constraints for which our propagator achieves bounds($\mathbb{Z}$) consistency or bounds($\mathbb{R}$) consistency only. In particular, Trick [24] presented a domain-consistent propagator for Linear$_=$ and Pesant [25] presented a domain-consistent propagator for L$_p$-Norm (including Spread and Deviation). Those propagators are based on dynamic programming ideas and have a higher time complexity than our approach.

11. Conclusion and future work

We have studied how to propagate pairs of Sum constraints that respect a discrete convexity condition. From this condition, we have derived a parametric propagator, which can be instantiated to be competitive with previously published propagators, often matching their time complexity, despite its generality.

There are a number of open questions we plan to address in the future. Can we *automatically* generate the instantiation of the parameters from the definitions of the f_i and g_i? Can we make an *incremental* propagator that has a better time complexity along a branch of the search tree? Can we extend the approach to functions that take more than one argument, say $f_i(x_i, y_i)$ for variables y_i distinct from each other, or $f_i(x_i, y)$ for a shared variable y, covering for instance the variable-average version of Spread [2,26]? Can we deal with more than two sum constraints in one propagator?

References

[1] J.-N. Monette, N. Beldiceanu, P. Flener, J. Pearson, A parametric propagator for discretely convex pairs of Sum constraints, in: C. Schulte (Ed.), CP 2013, in: LNCS, vol. 8124, Springer, 2013, pp. 529–544.
[2] G. Pesant, J.-C. Régin, SPREAD: a balancing constraint based on statistics, in: P. van Beek (Ed.), CP 2005, in: LNCS, vol. 3709, Springer, 2005, pp. 460–474.
[3] P. Schaus, Y. Deville, P. Dupont, Bound-consistent Deviation constraint, in: C. Bessière (Ed.), CP 2007, in: LNCS, vol. 4741, Springer, 2007, pp. 620–634.
[4] T. Petit, J.-C. Régin, N. Beldiceanu, A $\Theta(n)$ bound-consistency algorithm for the Increasing Sum constraint, in: J. Lee (Ed.), CP 2011, in: LNCS, vol. 6876, Springer, 2011, pp. 721–728.
[5] J.-F. Puget, Improved bound computation in presence of several Clique constraints, in: M. Wallace (Ed.), CP 2004, in: LNCS, vol. 3258, Springer, 2004, pp. 527–541.
[6] J.-C. Régin, T. Petit, The Objective Sum constraint, in: T. Achterberg, J.C. Beck (Eds.), CPAIOR 2011, in: LNCS, vol. 6697, Springer, 2011, pp. 190–195.
[7] C. Schulte, P.J. Stuckey, When do bounds and domain propagation lead to the same search space?, ACM Trans. Program. Lang. Syst. 27 (3) (2005) 388–425.
[8] C. Choi, W. Harvey, J. Lee, P. Stuckey, Finite domain bounds consistency revisited, in: A. Sattar, B.-h. Kang (Eds.), AI 2006: Advances in Artificial Intelligence, in: LNCS, vol. 4304, Springer, 2006, pp. 49–58.
[9] N. Razakarison, N. Beldiceanu, M. Carlsson, H. Simonis, GAC for a linear inequality and an Atleast constraint with an application to learning simple polynomials, in: SoCS 2013, AAAI Press, 2013, pp. 149–157.
[10] W. Harvey, J. Schimpf, Bounds consistency techniques for long Linear constraints, in: Proceedings of TRICS 2002, the Workshop on Techniques foR Implementing Constraint programming Systems, 2002, pp. 39–46.
[11] P. Schaus, J.-C. Régin, Bound-consistent Spread constraint, application to load balancing in nurse-to-patient assignments, EURO J. Comput. Optim. (2013) 1–24.
[12] K. Murota, Recent developments in discrete convex analysis, in: W. Cook, L. Lovász, J. Vygen (Eds.), Research Trends in Combinatorial Optimization, Springer, 2009, pp. 219–260.
[13] S. Fujishige, Submodular Functions and Optimization, 2nd edition, Annals of Discrete Mathematics, Elsevier, 2005.
[14] N. Beldiceanu, E. Contejean, Introducing global constraints in CHIP, Math. Comput. Model. 20 (12) (1994) 97–123.
[15] C. Bessière, E. Hebrard, B. Hnich, Z. Kızıltan, T. Walsh, Among, Common and Disjoint constraints, in: B. Hnich, M. Carlsson, F. Fages, F. Rossi (Eds.), CSCLP 2005; Revised Selected and Invited Papers, in: LNAI, vol. 3978, Springer-Verlag, 2006, pp. 28–43.
[16] OscaR Team, OscaR: Scala in OR, available from http://www.oscarlib.org, 2012.
[17] P. Schaus, Solving balancing and bin-packing problems with constraint programming, PhD Thesis, Université catholique de Louvain, Belgium, 2009.
[18] I.P. Gent, The recomputation manifesto, CoRR arXiv:1304.3674, available at http://arxiv.org/abs/1304.3674.
[19] T. Petit, N. Beldiceanu, X. Lorca, A generalized arc-consistency algorithm for a class of counting constraints, in: IJCAI 2011, AAAI Press, 2011, pp. 643–648, revised edition available at http://arxiv.org/abs/1110.4719.
[20] G. Katsirelos, N. Narodytska, T. Walsh, The SeqBin constraint revisited, in: M. Milano (Ed.), CP 2012, in: Lecture Notes in Computer Science, vol. 7514, Springer, 2012, pp. 332–347.
[21] J.-C. Régin, Cost-based arc consistency for global cardinality constraints, Constraints 7 (3–4) (2002) 387–405.
[22] Y. Deville, O. Barette, P. Van Hentenryck, Constraint satisfaction over connected row-convex constraints, Artif. Intell. 109 (1–2) (1999) 243–271.
[23] Y. Zhang, E.C. Freuder, Properties of tree convex constraints, Artif. Intell. 172 (12–13) (2008) 1605–1612.
[24] M.A. Trick, A dynamic programming approach for consistency and propagation for knapsack constraints, Ann. Oper. Res. 118 (1–4) (2003) 73–84.
[25] G. Pesant, Achieving domain consistency and counting solutions for dispersion constraints, INFORMS J. Comput. 27 (4) (2015) 690–703.
[26] S.C. Loong, W.-Y. Ku, J.C. Beck, $\mathbb{Q}$-bounds consistency for the spread constraint with variable mean, Constraints (2016) 1–7.