INFORMS JOURNAL ON COMPUTING
Vol. 27, No. 4, Fall 2015, pp. 690–703
ISSN 0899-1499 | EISSN 1526-5528 | 00 | 0000 | 0001

INFORMS
DOI 10.1287/ijoc.2015.0654

Achieving Domain Consistency and Counting Solutions for Dispersion Constraints *

Gilles Pesant
École Polytechnique de Montréal, Montreal, Canada
CIRRELT, Université de Montréal, Montreal, Canada
Gilles.Pesant@cirrelt.ca

Many combinatorial problems require of their solutions that they achieve a certain balance of given features. For this important aspect of modeling, the `spread` and `deviation` constraints have been proposed in Constraint Programming to express balance among a set of variables by constraining their mean and their overall deviation from the mean. Currently the only practical filtering algorithms known for these constraints achieve bounds consistency. In this paper we improve that filtering by presenting an efficient domain consistency algorithm that applies to both constraints. We also extend it to count solutions so that it can be used in counting-based search, a generic and effective family of branching heuristics that free the user from having to write problem-specific search heuristics. We provide a time complexity analysis of our contributions and also evaluate them empirically on benchmark problems.

Key words: combinatorial optimization, constraint programming, balance, spread constraint, deviation constraint, filtering algorithms, branching heuristics, counting-based search

1. Introduction

Many combinatorial problems require of their solutions that they achieve a certain balance of given features. For example in assembly line balancing the workload of the line operators must be balanced (Falkenauer 2005). In rostering we usually talk of fairness instead of balance, because of the human factor — here we want a fair distribution of weekends off or of night shifts among workers, for example (Pesant 2008). Schaus et al. (2009) aim for an even distribution of nurses' workload in a neonatal care unit. Lemaître et al. (1999) use an earth observation satellite scheduling and sharing problem to investigate different ways of handling fairness among agents with competing observation requirements while maximizing a weighted sum of the observations made. Take in particular the Balanced

* A preliminary version of this work was presented during the Tenth International Workshop on Constraint Modeling and Reformulation held in Perugia, Italy in September 2011.

Academic Curriculum Problem (Problem 30 of Gent and Walsh (1999)) in which courses are assigned to semesters so as to balance the academic load between semesters. Because of additional constraints (prerequisite courses, minimum and maximum number of courses per semester) and a varying number of credits per course, reaching perfect balance is generally impossible. A common way of encouraging balance at the modeling level is to set reasonable bounds on each load, tolerating a certain deviation from the ideal value, but it has the disadvantage of putting on an equal footing solutions with quite different balance. Consider assigning courses totalling 60 credits among six semesters, requiring that loads belong to the interval $[8, 9, \ldots, 12]$, and aiming for an ideal load of 10: sets of loads $\{10, 10, 10, 10, 9, 11\}$ and $\{8, 8, 8, 12, 12, 12\}$ both satisfy the restriction but the former is much more balanced. Another way is to minimize some combination of the individual deviations from the ideal value, but if other criteria to optimize are already present one must come up with suitable weights for the different terms of the objective function.

Yet another way is to bound the sum of individual deviations as a constraint: the `spread` (Pesant and Régin 2005) and `deviation` (Schaus et al. 2007b) constraints express balance among a set of variables in a Constraint Programming (CP) model by constraining their mean and their overall deviation from the mean. In order to formally describe these constraints we first present some basic definitions about CP. Constraint programming solves combinatorial problems by actively using the constraints of the problem to implicitly eliminate infeasible regions of the solution space. The algorithm at the heart of CP, termed *constraint propagation*, implements complex logical reasoning over the set of constraints by combining local inference on individual constraints.

Definition 1 (finite-domain and bounded-domain variables). A *finite-domain (discrete) variable* takes a value in a finite set called its *domain*. A *bounded-domain (continuous) variable* takes a value in a closed real interval also called its domain.

Definition 2 (Constraint Satisfaction Problem (X, D, C)). Given a finite set of variables, $X = \{x_1, \ldots, x_n\}$, a domain for each variable, $D = \{D_1, \ldots, D_n\}$, $x_i \in D_i$, and a finite set of constraints (relations) on subsets of the variables, $C = \{c_1, \ldots, c_m\}$ where $c_j(x_1, \ldots, x_k) \subseteq D_1 \times \cdots \times D_k$, find an assignment of values to X from D such that each constraint in C is satisfied (i.e. the relation contains the tuple of values).

We will use the following notation: $d = \max_{i=1}^{n} |D_i|$, $D_i^{\max} = \max\{v \in D_i\}$, $D_i^{\min} = \min\{v \in D_i\}$, $D^{\max} = \max_{i=1}^{n} D_i^{\max}$, $D^{\min} = \min_{i=1}^{n} D_i^{\min}$, and $[D] = D^{\max} - D^{\min}$.

The result of the local inference on constraints, filtering out some inconsistent values in the domain of individual variables, is characterized formally as a particular level of consistency being achieved.

DEFINITION 3 (DOMAIN CONSISTENCY). A constraint $c(x_1,\ldots,x_k)$ is *domain consistent* if and only if for every $i \in \{1,\ldots,k\}$ and every $v \in D_i$

$$\forall j \in \{1,\ldots,k\} \setminus \{i\} \quad \exists v_j \in D_j \text{ such that}$$

$$c(v_1,\ldots,v_{i-1},v,v_{i+1},\ldots,v_k) \text{ is satisfied.}$$

We allow ourselves to restrict this characterization to a subset of the variables $\{x_i \in X : i \in S \subset \{1,\ldots,k\}\}$ by saying that $c(x_1,\ldots,x_k)$ is *domain consistent on* S if i ranges over S in the previous definition.

DEFINITION 4 (BOUNDS CONSISTENCY). A constraint $c(x_1,\ldots,x_k)$ is *bounds*(D) *consistent* if and only if for every $i \in \{1,\ldots,k\}$

$$\forall j \in \{1,\ldots,k\} \setminus \{i\} \quad \exists v_j \in D_j \text{ such that}$$

$$c(v_1,\ldots,v_{i-1},D_i^{\min},v_{i+1},\ldots,v_k) \text{ and } c(v_1,\ldots,v_{i-1},D_i^{\max},v_{i+1},\ldots,v_k) \text{ are satisfied.}$$

A constraint $c(x_1,\ldots,x_k)$ is *bounds*$(\mathbb{Z})$ *consistent* if and only if for every $i \in \{1,\ldots,k\}$

$$\forall j \in \{1,\ldots,k\} \setminus \{i\} \quad \exists v_j \in \{D_j^{\min}, D_j^{\min}+1,\ldots,D_j^{\max}\} \text{ such that}$$

$$c(v_1,\ldots,v_{i-1},D_i^{\min},v_{i+1},\ldots,v_k) \text{ and } c(v_1,\ldots,v_{i-1},D_i^{\max},v_{i+1},\ldots,v_k) \text{ are satisfied.}$$

A constraint $c(x_1,\ldots,x_k)$ is *bounds*$(\mathbb{R})$ *consistent* if and only if for every $i \in \{1,\ldots,k\}$

$$\forall j \in \{1,\ldots,k\} \setminus \{i\} \quad \exists v_j \in \text{ real interval } [D_j^{\min}, D_j^{\max}] \text{ such that}$$

$$c(v_1,\ldots,v_{i-1},D_i^{\min},v_{i+1},\ldots,v_k) \text{ and } c(v_1,\ldots,v_{i-1},D_i^{\max},v_{i+1},\ldots,v_k) \text{ are satisfied.}$$

Here as well we allow ourselves to restrict this characterization to a subset of the variables $\{x_i \in X : i \in S \subset \{1,\ldots,k\}\}$ by saying that $c(x_1,\ldots,x_k)$ is *bounds*$(D$ *or* $\mathbb{Z}$ *or* $\mathbb{R})$ *consistent on* S if i ranges over S in the previous definitions.

Note that domain consistency is a stronger property (i.e. may filter out more values) than bounds(D) consistency, itself stronger than bounds($\mathbb{Z}$) consistency, which is stronger than bounds($\mathbb{R}$) consistency. Typically this inference process is applied at every node of a depth-first-search tree in which branching corresponds to fixing a variable to a value in its domain, thus triggering more constraint propagation. We call *variable-selection heuristic* and *value-selection heuristic* the way one decides which variable to branch on and which value to try first, respectively.

We now formally define the constraints previously introduced to express balance.

Definition 5 (spread constraint (Pesant and Régin 2005)). Given a set of finite-domain integer variables $X = \{x_1, x_2, \ldots, x_n\}$, bounded-domain variables μ, σ, and finite-domain variable $\tilde{x}$, constraint $\texttt{spread}(X, \mu, \sigma, \tilde{x})$ states that the collection of values taken by the variables in X exhibits an arithmetic mean μ, a standard deviation σ, and a median $\tilde{x}$.

This constraint measures balance using the standard deviation, or root-mean-square, on the values taken by the variables. Note that the mean does not have to be fixed. Even though the constraint also allows constraining the median, we will not consider that feature in this paper. Pesant and Régin (2005) achieve bounds($\mathbb{R}$) consistency on X in $\mathcal{O}(n^2)$ time for `spread`. For a fixed mean Schaus and Régin (2013) improve on this by achieving bounds($\mathbb{Z}$) consistency on X in $\mathcal{O}(n \log n)$ time.

Definition 6 (deviation constraint (Schaus et al. 2007b)). Given a set of finite-domain integer variables $X = \{x_1, x_2, \ldots, x_n\}$, a *fixed* value μ, and bounded-domain variable s, constraint $\texttt{deviation}(X, \mu, s)$ states that the collection of values taken by the variables of X exhibits an arithmetic mean μ and a sum of absolute differences to μ, $\sum_{i=1}^{n} |x_i - \mu|$, of s.

For this other constraint an analytically simpler expression of balance is used and the mean is considered fixed, both features contributing to achieve bounds($\mathbb{Z}$) consistency on X in $\mathcal{O}(n)$ time (Schaus et al. 2007a). A simpler algorithm with the same characteristics was initially proposed by Schaus et al. (2007b) for the case of a fixed *integral* mean. More recently Monette et al. (2013) describe a propagator for certain pairs of `sum` constraints, of which `spread` and `deviation` are special cases. It achieves bounds($\mathbb{Z}$) consistency on X for the latter in $\mathcal{O}(n)$ time and for the former (as well as for higher norms) in $\mathcal{O}(n \cdot |\cup_{i=1}^{n} D_i|)$ time. It can also be used to achieve bounds($\mathbb{Z}$) consistency on deviation variable Δ (defined below).

DEFINITION 7 (L_p-DISTANCE, L_p-DEVIATION). The *L_p-distance* between two n-dimensional points $\mathbf{x}$ and $\mathbf{y}$ is given by the p-norm $\|\mathbf{x}-\mathbf{y}\|_p = (\sum_{i=1}^{n}|x_i-y_i|^p)^{1/p}$. We introduce the related concept of L_p-deviation in which $\mathbf{y}=\boldsymbol{\mu}$ and we drop the p^{th} root:

$$\sum_{i=1}^{n}|x_i-\mu|^p.$$

Special cases of interest here are the L_1- and L_2-deviations which respectively relate to the `deviation` and `spread` constraints. We introduce `dispersion` constraints to represent both previous constraints.

DEFINITION 8 (DISPERSION CONSTRAINT). Given a set of finite-domain integer variables $X=\{x_1,x_2,\ldots,x_n\}$, bounded-domain variables μ and Δ, and natural number p, constraint `dispersion`(X,μ,Δ,p) states that the collection of values taken by the variables of X exhibits an arithmetic mean μ and an L_p-deviation Δ.

No L_p-deviation dominates the others in terms of better balance — which one to use is application-dependent (see e.g. Schaus et al. (2009)). But the difference between a domain-consistent and a bounds-consistent `dispersion` constraint can be more values filtered from domains and ultimately a smaller search tree.

EXAMPLE 1. Consider a `dispersion` constraint on variables $x_1\in\{8,9,10\}$, $x_2\in\{10,13\}$, and $x_3\in\{8,10,12\}$ whose arithmetic mean is fixed to 10. There are three instantiations achieving that mean, $\{\langle 8,10,12\rangle,\langle 9,13,8\rangle,\langle 10,10,10\rangle\}$, and every value of every domain is supported. According to the L_1-deviation, these instantiations get a Δ value of 4, 6, and 0 respectively. For the L_2-deviation, the Δ values are 8, 14, and 0. Our constraint will typically bound the amount of deviation from above — call β the maximum value for Δ: if $4\leq\beta<6$ with the L_1-deviation, the middle value 9 should be removed from the domain of x_1; the same can be said if $8\leq\beta<14$ with the L_2-deviation. A bounds-consistent filtering algorithm will not make that deletion but a domain-consistent one will.

As Proposition 1 of Schaus et al. (2007b) states, achieving domain consistency is $\mathcal{NP}$-hard for `spread` and `deviation`. Note however that it is not $\mathcal{NP}$-hard *in the strong sense* so a pseudo-polynomial time algorithm is not ruled out.

In this paper we exploit this fact and propose a new filtering algorithm that can be used for `dispersion` constraints (and thus both for `spread` (disregarding the median) and `deviation`), that achieves domain consistency, and whose time complexity is similar in

practice to that of the previously proposed algorithms (Section 2). We also extend that algorithm to provide counting information that is used in counting-based search heuristics (Zanarini and Pesant 2009) (Section 3). Both contributions are then evaluated empirically on the Balanced Academic Curriculum Problem, the Nurse to Patient Assignment Problem, and on random instances (Section 4).

2. Domain Consistency for Dispersion Constraints

For the moment, and to help clarity, we consider a fixed mean μ but we will discuss a variable mean in Section 2.2.

2.1. Fixed Mean

The first part of such constraints is formalized as

$$\sum_{i=1}^{n} x_i = n\mu. \tag{1}$$

About the second part, the deviation Δ, for `spread` we transfer the square root and the n denominator over to the right-hand side in order to make it correspond to an L_2-deviation,

$$\sum_{i=1}^{n} (x_i - \mu)^2 = \Delta \tag{2}$$

and we leave it as is for `deviation` (L_1-deviation),

$$\sum_{i=1}^{n} |x_i - \mu| = \Delta. \tag{3}$$

Linear arithmetic constraints, $\ell \leq \sum_{i=1}^{n} c_i x_i \leq u$, frequently arise in CP models. One often maintains bounds($\mathbb{R}$) consistency on such constraints because domain consistency can be too costly: Trick (2003) proposed a dynamic programming algorithm achieving domain consistency for knapsack constraints whose time complexity is in $\mathcal{O}(ndu)$: nu is an upper bound on the number of states in the recursion and d is an upper bound on the number of terms to consider in the recursive formula. Because it is linearly related to the size of the upper bound, it is a pseudo-polynomial time algorithm and hence possibly too slow when u is large. Equation 1 is a special case of this in which variables have unit coefficients and $\ell = u$. In this section we show that for this special case and in the context of `dispersion` constraints, that algorithm essentially runs in polynomial time and can be extended to keep track of deviations as well by maintaining cumulative information at each state as is

commonly done in dynamic programming, and particularly in CP for the `cost-regular` constraint (Demassey et al. 2006).

We define the individual deviation of value v with respect to the mean as $\delta_\mu(v) = |v - \mu|$ for the L_1-deviation and $\delta_\mu(v) = (v-\mu)^2$ for the L_2-deviation. Since what follows is independent of the L_p-deviation being used, we will often drop the last argument of the constraint and simply write $\texttt{dispersion}(X, \mu, \Delta)$.

Let $f(i,j)$ represent the smallest cumulative deviation $\sum_{k=1}^{i} \delta_\mu(x_k)$ achievable by $\{x_1, x_2, \ldots, x_i\}$ such that $\sum_{k=1}^{i}(x_k - \lfloor\mu\rfloor) = j$. We use the integer part of μ here because we wish for j to be an integer. It can be computed recursively as

$$f(i,j) = \min_{v \in D_i}\{\delta_\mu(v) + f(i-1, j-(v-\lfloor\mu\rfloor))\} \quad 1 \le i \le n,\ \underline{j} \le j \le \overline{j} \tag{4}$$

$$f(0,j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases} \tag{5}$$

Its first index ranges over the number of variables; the range of the second index can be bounded, as we shall see in the proof of Theorem 2 ($\underline{j}$ and $\overline{j}$ are defined later in Algorithm 2).

LEMMA 1. *Equations 4 and 5 correctly define $f(i,j)$.*

Proof We show the correctness of the recurrence by induction on i. At the base case ($i = 0$), both summations are empty so $j = 0$ and the deviation is null. As our induction hypothesis, for every j, let

$$f(i-1,j) = \min \sum_{k=1}^{i-1} \delta_\mu(x_k) \text{ such that } \sum_{k=1}^{i-1}(x_k - \lfloor\mu\rfloor) = j \text{ and } x_k \in D_k\ \forall 1 \le k \le i-1.$$

Then by definition

$$f(i,j) = \min_{v \in D_i}\{\delta_\mu(v) + f(i-1, j-(v-\lfloor\mu\rfloor))\}$$

and using the induction hypothesis we get

$$f(i,j) = \min_{v \in D_i}\{\delta_\mu(v) + \min \sum_{k=1}^{i-1} \delta_\mu(x_k)\}$$

such that $\sum_{k=1}^{i-1}(x_k - \lfloor\mu\rfloor) = j - (v - \lfloor\mu\rfloor)$, which we rewrite as $\sum_{k=1}^{i}(x_k - \lfloor\mu\rfloor) = j$ with $x_i = v$. And so

$$f(i,j) = \min \sum_{k=1}^{i} \delta_\mu(x_k) \text{ such that } \sum_{k=1}^{i}(x_k - \lfloor\mu\rfloor) = j,$$

as desired. □

Similarly we define f', representing the *largest* cumulative deviation. Arguably most of the time one wishes to constrain the cumulative deviation from above in order to achieve balance. Sometimes one may also wish to constrain it from below to obtain some diversity. Rarely would we want both at the same time. The next theorem covers these two important cases.

THEOREM 1. *We distinguish two cases.*

i. $\Delta \in [0, \Delta^{\max}]$*:*

`dispersion`(X, μ, Δ) *is satisfiable iff* $f(n, n(\mu - \lfloor\mu\rfloor)) \leq \Delta^{\max}$

ii. $\Delta \in [\Delta^{\min}, \infty)$*:*

`dispersion`(X, μ, Δ) *is satisfiable iff* $f'(n, n(\mu - \lfloor\mu\rfloor)) \geq \Delta^{\min}$

Proof At $f(n, n(\mu - \lfloor\mu\rfloor))$ and $f'(n, n(\mu - \lfloor\mu\rfloor))$ we have $\sum_{k=1}^{n}(x_k - \lfloor\mu\rfloor) = n(\mu - \lfloor\mu\rfloor)$ which we rewrite as $\sum_{k=1}^{n} x_k - n\lfloor\mu\rfloor = n\mu - n\lfloor\mu\rfloor$ or $\sum_{k=1}^{n} x_k = n\mu$ so it is equivalent to Equation 1. Now consider the first case. If $f(n, n(\mu - \lfloor\mu\rfloor)) \leq \Delta^{\max}$ (and it is non-negative by definition) then the corresponding assignment satisfies the `balance` constraint. Conversely if the constraint is satisfied then there must exist an assignment of cumulative deviation at most $\Delta^{\max}$, so the smallest cumulative deviation $f(n, n(\mu - \lfloor\mu\rfloor))$ certainly is at most that. The second case proceeds similarly. If $f'(n, n(\mu - \lfloor\mu\rfloor)) \geq \Delta^{\min}$ then the corresponding assignment satisfies the `dispersion` constraint. Conversely if the constraint is satisfied then there must exist an assignment of cumulative deviation at least $\Delta^{\min}$, so the largest cumulative deviation $f'(n, n(\mu - \lfloor\mu\rfloor))$ certainly is at least that. □

Note that for the special case of an integral mean, $n(\mu - \lfloor\mu\rfloor) = 0$. Besides being useful to decide satisfiability, f and f' can tighten the lower and upper bounds on Δ, respectively, achieving bounds(D) consistency on it. Algorithm 1 describes how to compute f. We can view the process as building a layered graph such as Figure 1, in which vertices are the elements of set S (growing progressively as we reach from one value of i to the next) and edges are the transitions considered in the innermost loop. The top and bottom dashed lines originating from state $(0, 0)$, with respective slopes $D^{\max} - \lfloor\mu\rfloor$ and $D^{\min} - \lfloor\mu\rfloor$, define the area in which vertices may occur when reaching from $(0, 0)$. Similar dashed lines originate from state $(n, n(\mu - \lfloor\mu\rfloor))$, parallel to the first two, defining the area in which vertices may reach $(n, n(\mu - \lfloor\mu\rfloor))$. The test at Line 6, based on the latter lines, avoids adding to S states that cannot possibly be extended to the final state $(n, n(\mu - \lfloor\mu\rfloor))$.

Algorithm 1: Computing recursive function f

1 $f(0,0) \leftarrow 0$;
2 $S \leftarrow \{(0,0)\}$;
3 **for** $i = 1, 2, \ldots, n$ **do**
4 **forall the** $(i-1, j) \in S$ **do**
5 **forall the** $v \in D_i$ **do**
6 **if** $(n-i) \cdot (\lfloor\mu\rfloor - D^{\max}) \leq (j + v - \lfloor\mu\rfloor) - n(\mu - \lfloor\mu\rfloor) \leq (n-i) \cdot (D^{\min} - \lfloor\mu\rfloor)$ **then**
7 **if** $(i, j + v - \lfloor\mu\rfloor) \notin S$ **then**
8 $f(i, j + v - \lfloor\mu\rfloor) \leftarrow \delta_\mu(v) + f(i-1, j)$;
9 $S \leftarrow S \cup \{(i, j + v - \lfloor\mu\rfloor)\}$;
10 **else**
11 $f(i, j + v - \lfloor\mu\rfloor) \leftarrow \min(f(i, j + v - \lfloor\mu\rfloor), \delta_\mu(v) + f(i-1, j))$;
12 **return** f;

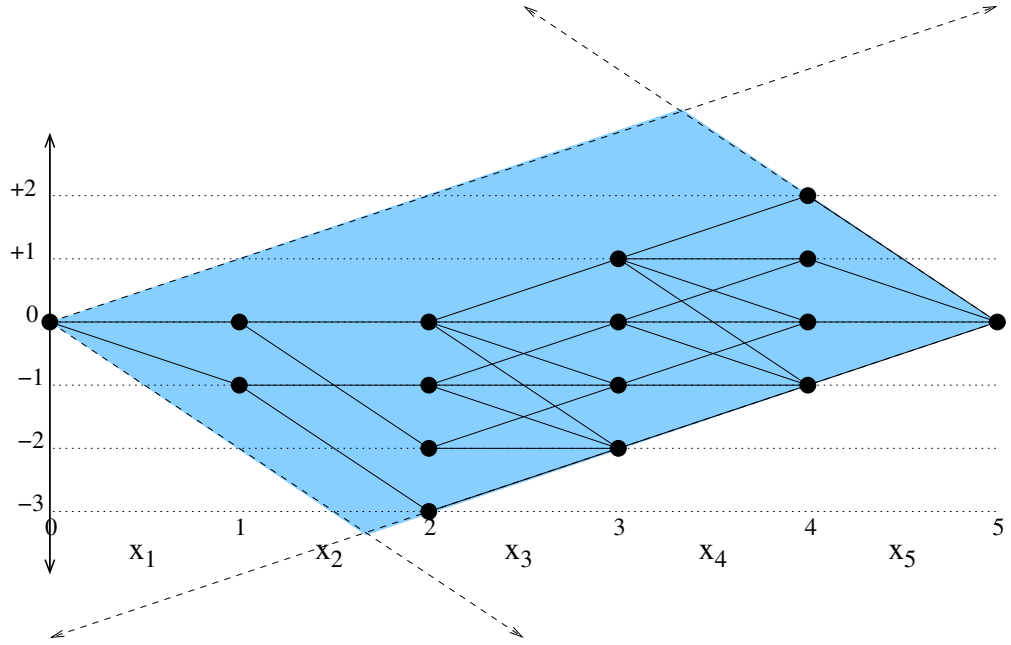

Figure 1 **An example of recursion graph for $n = 5$ and an integral μ.**

THEOREM 2. *Algorithm 1 runs in $\mathcal{O}(n^2 d[D])$ time.*

Proof Lines 6-11 inside the loops can be done in $\mathcal{O}(1)$. The loop at Line 3 clearly requires n iterations and the one at Line 5 requires at most d. The middle loop at Line 4

deserves a closer analysis. For a given i, how many states can be in S? From the previous discussion, states are restricted to the shaded area of Figure 1. The maximum number of states of any given abscissa i is related to the vertical distance between the parallel sides, given in particular by the difference between their respective y-intercepts. For the two sides of slope $D^{\max} - \lfloor\mu\rfloor$, the y-intercepts are 0 and $b = y - mx = n(\mu - \lfloor\mu\rfloor) - (D^{\max} - \lfloor\mu\rfloor)n = n(\mu - D^{\max})$. For the other pair, the y-intercepts are 0 and $n(\mu - D^{\min})$. We take the minimum of the two distances since they jointly constrain the range of states of a given abscissa:

$$\begin{aligned}\min(0 - n(\mu - D^{\max}), n(\mu - D^{\min}) - 0) + 1 &= n\min(D^{\max} - \mu, \mu - D^{\min}) + 1\\ &\leq n(D^{\max} - D^{\min})/2 + 1\\ &\leq n[D] + 1\end{aligned}$$

The result follows. □

COROLLARY 1. *Bounds(D) consistency on Δ can be achieved in $\mathcal{O}(n^2 d[D])$ time for the* `dispersion` *constraint (and hence also for* `spread` *and* `deviation`*) with fixed mean.*

Note that this is a stronger level of consistency than that achieved in Monette et al. (2013).

But we are even more interested in filtering the domains of the x_i variables and to this end, we also define the backward version of f, $b(i,j)$ representing the smallest cumulative deviation achievable by $\{x_{i+1}, x_{i+2}, \ldots, x_n\}$ such that $\sum_{k=i+1}^{n}(x_k - \lfloor\mu\rfloor) = j$. (As before we define b' similarly.)

$$b(i,j) = \min_{v \in D_{i+1}} \{\delta_\mu(v) + b(i+1, j - (v - \lfloor\mu\rfloor))\} \quad 0 \leq i \leq n-1,\ \underline{j} \leq j \leq \overline{j} \tag{6}$$

$$b(n,j) = \begin{cases} 0, & \text{if } j = n(\mu - \lfloor\mu\rfloor) \\ -\infty, & \text{otherwise} \end{cases} \tag{7}$$

The smallest deviation possible for a solution containing instantiation $x_i = v$ is the minimum of expression $\{f(i-1,j) + \delta_\mu(v) + b(i, j + (v - \lfloor\mu\rfloor))\}$ over j. Algorithm 2 describes how to achieve domain consistency on X in the case $\Delta \in [0, \Delta^{\max}]$. Most of it is concerned with computing for each i the possible range $[\underline{j}, \underline{j}+1, \ldots, \overline{j}]$ for j. The inner loop then tests each domain value for removal. For the case $\Delta \in [\Delta^{\min}, \infty)$, we are seeking the largest

deviation possible so Line 1 would compute f' and b' instead and the condition on Line 14 would be replaced by $\max_{j=\underline{j}}^{\overline{j}}\{f'(i-1,j)+\delta_\mu(v)+b'(i,j+(v-\lfloor\mu\rfloor))\}<\Delta^{\min}$.

Algorithm 2: Domain consistency algorithm for the `dispersion` constraint

1 Compute f and b;
2 $\underline{fwd}\leftarrow 0$;
3 $\underline{bwd}\leftarrow \sum_{i=1}^{n}(D_i^{\max}-\lfloor\mu\rfloor)$;
4 $\overline{fwd}\leftarrow 0$;
5 $\overline{bwd}\leftarrow \sum_{i=1}^{n}(\lfloor\mu\rfloor-D_i^{\min})$;
6 **for** $i=1,2,\ldots,n$ **do**
7 $\quad \underline{fwd}\leftarrow \underline{fwd}+(\lfloor\mu\rfloor-D_i^{\min})$;
8 $\quad \underline{bwd}\leftarrow \underline{bwd}-(D_i^{\max}-\lfloor\mu\rfloor)$;
9 $\quad \overline{fwd}\leftarrow \overline{fwd}+(D_i^{\max}-\lfloor\mu\rfloor)$;
10 $\quad \overline{bwd}\leftarrow \overline{bwd}-(\lfloor\mu\rfloor-D_i^{\min})$;
11 $\quad \underline{j}\leftarrow \max(\underline{fwd},\underline{bwd})$;
12 $\quad \overline{j}\leftarrow \min(\overline{fwd},\overline{bwd})$;
13 $\quad$ **forall the** $v\in D_i$ **do**
14 $\quad\quad$ **if** $\min_{j=\underline{j}}^{\overline{j}}\{f(i-1,j)+\delta_\mu(v)+b(i,j+(v-\lfloor\mu\rfloor))\}>\Delta^{\max}$ **then**
15 $\quad\quad\quad$ Remove value v from domain D_i;

THEOREM 3. *Domain consistency on X can be achieved in $\mathcal{O}(n^2d[D])$ time for the* `dispersion` *constraint (and hence also for* `spread` *and* `deviation`*) with fixed mean.*

Proof We analyze Algorithm 2. The computation of f and b (or f' and b') was already shown to take $\mathcal{O}(n^2d[D])$ time. The body of the nested loops will be executed at most nd times and each execution essentially finds the minimum of $\overline{j}-\underline{j}+1$ terms, that number being in $\mathcal{O}(n[D])$ as we saw in the proof of Theorem 2. □

In practice for the most likely case $\Delta\in[0,\Delta^{\max}]$, d and $[D]$ are small if μ is fixed since each x_i is meant to be close to μ. Consequently the complexity of the filtering algorithm is essentially quadratic in the number of variables. Because the graph structure is maintained, the algorithm is also incremental and the time complexity of re-establishing domain

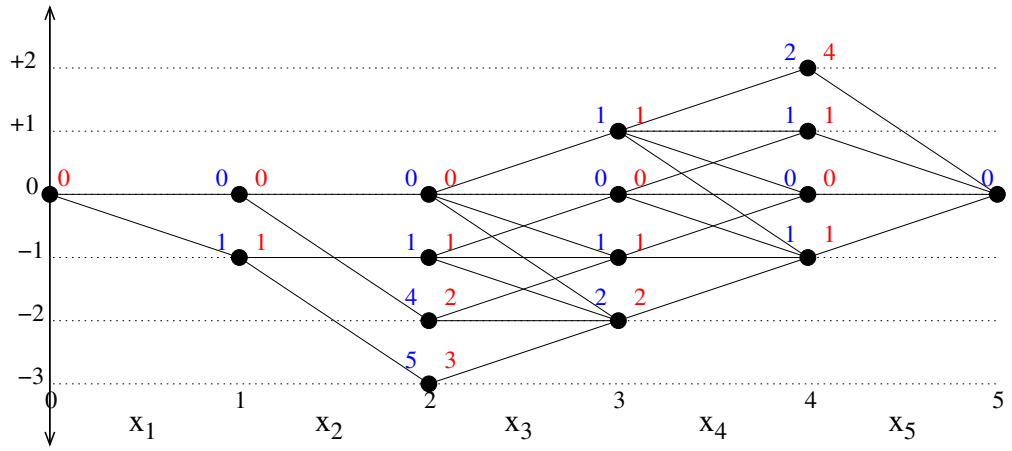

Figure 2 **Layered graph for Example 2 with f and b values on the left and right of each vertex, respectively.**

consistency upon a domain change is proportional to the number of necessary updates in the graph.

EXAMPLE 2. Consider variables $x_1 \in \{10, 11\}$, $x_2 \in \{9, 11\}$, and $x_3, x_4, x_5 \in \{9, 10, 11, 12\}$ on which a `dispersion` constraint using the L_2-deviation (i.e. `spread`) is stated with $\mu = 11$: the corresponding graph is shown in Figure 2. Constrained by $\Delta \leq 5$, value 9 for x_3 can be removed since $f(2,0) + \delta_{11}(9) + b(3,-2) = 0 + 4 + 2 = 6$. In fact value 9 would be removed from the domain of x_2, x_4, and x_5 as well.

Note that the preceding treatment could be applied to other deviations as well. For example to work with the L_0-deviation (number of deviations) the individual deviation (δ_μ) becomes a 0-1 function, and for the L_∞-deviation (maximum deviation) we would take the maximum of the two terms in (4) and (6) instead of the sum.

2.2. Variable Mean

We end this section by addressing the issue of a fixed mean. In most contexts we have a known number of features to distribute as evenly as possible among a set of "recipients": the mean is therefore fixed since it corresponds to the ratio of the number of features to the number of recipients. This situation occurs in rostering, for example, when night shifts should be evenly distributed among staff members. Contrast this with the handling of *broken weekends* (a weekend on which one day is worked and the other not), a generally undesirable feature: we often do not know in advance how many such weekends will occur in a schedule but we nevertheless wish the number of broken weekends to be evenly distributed among all staff members. In such a context we cannot fix the mean a priori, so being able to work with an unspecified mean is worthwhile.

Our algorithm can be generalized to the wider context of a variable mean, but at a computational cost. Essentially we need to add a third dimension to our recurrences, corresponding to the span of allowed values for μ. A similar graph is built independently for every allowed value for μ since its definition, both in a vertex's second dimension and in its deviation label, depends on μ. For convenience we can think of these allowed values as integers once we scale them by n. So the size of the data structure and the computation time are increased by a $n\mu^{\max} - n\mu^{\min} + 1$ factor.

COROLLARY 2. *Domain consistency on X can be achieved in $\mathcal{O}(n^3 d[D](\mu^{\max} - \mu^{\min}))$ time for the* `dispersion` *constraint with variable mean.*

3. Counting the Solutions to a Dispersion Constraint

The filtering algorithm of the previous section maintains a data structure that represents every solution to the constraint: a solution is a path in the layered graph whose deviation labels on the arcs sum up to Δ. By augmenting the data structure with multiple labels on each vertex, representing not only a path of smallest or largest deviation but every path, we can count the number of solutions. We limit ourselves to the more common case in which the deviation is upper-bounded. We first recall some definitions from Zanarini and Pesant (2009).

Given a constraint γ defined on the set of variables $\{x_1, \ldots, x_k\}$ and respective finite domains $D_{i\ 1 \le i \le k}$, let $\#\gamma(x_1, \ldots, x_k)$ denote the number of solutions to constraint γ. Given a variable x_i in the scope of γ, and a value $v \in D_i$, we call

$$\sigma(x_i, v, \gamma) = \frac{\#\gamma(x_1, \ldots, x_{i-1}, v, x_{i+1}, \ldots, x_k)}{\#\gamma(x_1, \ldots, x_k)}$$

the *solution density* of pair (x_i, v) in γ. It measures how often a certain assignment is part of a solution of the constraint γ.

Intuitively one would expect that for a `dispersion` constraint bounding deviation from above, the value $v \in D_i$ reaching the highest solution density for x_i would be one minimizing $\delta_\mu(v)$ since it leaves more freedom to the remaining variables. The corresponding value-ordering heuristic would then be trivial to compute. However the following example shows that it is not necessarily the case.

EXAMPLE 3. Consider again variables $x_1 \in \{10, 11\}$, $x_2 \in \{9, 11\}$, $x_3, x_4, x_5 \in \{9, 10, 11, 12\}$, and $\mu = 11$. Table 1 gives the solution densities for x_3 under a few upper

Table 1 Solution densities for x_3 in Example 3.

value	max L_1-deviation			max L_2-deviation		
	2	4	8	2	4	8
9	0	0,05	0,05	0	0	0,05
10	0,20	0,16	0,15	0,20	0,23	0,15
11	**0,50**	0,32	0,30	**0,50**	**0,38**	0,30
12	0,30	**0,47**	**0,50**	0,30	**0,38**	**0,50**

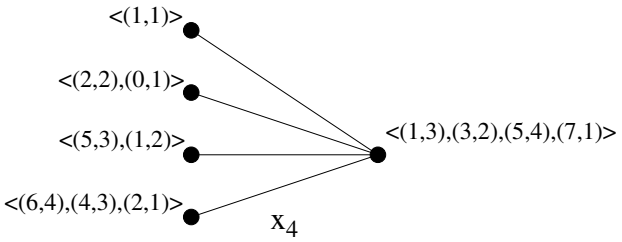

Figure 3 A detailed view of part of the graph at Figure 2, showing the number of incoming and outgoing paths of each amount of deviation.

bounds for the L_1 and L_2-deviations. The value with maximum solution density in each case is shown in bold. We can observe that the resulting density is influenced both by the stringency of the deviation bound and by the particular distribution of available values, which may pull in opposite directions.

Beyond the qualitative preoccupation of ranking values, solution densities offer the additional advantage of quantifying values relative to one another. This is important for example if one wishes to choose nondeterministically between closely ranked values but with a bias according to quantitative information. In the context of counting-based search, it offers a normalized score that is more easily compared with similar information originating from the other constraints in the model.

For the counting algorithm, we replace the single-value labels at the left and right of each vertex in the layered graph by lists of pairs of values (Figure 3). The list on the left of vertex (i,j) is denoted $\langle(\Delta^\ell_1(i,j),\#^\ell_1(i,j)),\ldots,(\Delta^\ell_{m^\ell_{ij}}(i,j),\#^\ell_{m^\ell_{ij}}(i,j))\rangle$. Every possible cumulative deviation $\Delta^\ell_k(i,j)$ achieved by some assignment of $x_1,\ldots,x_i$ with $\sum_{k=1}^{i}(x_k-\mu)=j$ is represented in the list, which is in strict decreasing order of deviation. Each $\#^\ell_k(i,j)$ reports the number of partial assignments achieving a deviation of *at most* $\Delta^\ell_k(i,j)$ (therefore these decrease in value). The list on the right of vertex (i,j), $\langle(\Delta^r_1(i,j),\#^r_1(i,j)),\ldots,(\Delta^r_{m^r_{ij}}(i,j),\#^r_{m^r_{ij}}(i,j))\rangle$, represents similar information except that it is in strict *increasing* order of deviation and that $\#^r_k(i,j)$ reports the number of partial assignments achieving a deviation of *exactly* $\Delta^r_k(i,j)$. As we will see in Algorithm 4, this difference in the structure

of the left and right lists allows a more efficient combination of counting information when computing solution densities. Note that the total number of solutions to the constraint is given by $\#_1^\ell(n, n(\mu - \lfloor\mu\rfloor))$.

Algorithm 3 computes the left labels on vertices of the layered graph. The loop starting at Line 5 iterates through the incoming arcs of vertex (i, j). The loop at Line 7 first goes down the left labels of the origin of the incoming arc until a cumulative deviation from the source respecting the upper bound is found. The loop starting at Line 10 then processes the remaining labels, either adding their number of paths to an existing entry or creating a new entry at the appropriate place in the ordered list L representing the labels for vertex (i, j). (Lines 21-24 complete the list once either one of the input lists have been exhausted.) Finally the loop at Line 27 traverses L backwards to compute the prefix sums required by our data structure. Right labels are computed similarly but without that last loop.

Algorithm 4 returns the solution density of variable-value pair (x_i, v). The loop starting at Line 2 iterates through the arcs of the graph which correspond to using value v for variable x_i. The left and right labels at the respective endpoints of each arc are traversed to find feasible combinations: at Line 4 increasing amounts of deviation from the mean are considered from the right; Lines 5-6 find the largest amount of deviation possible from the left without exceeding the upper bound on cumulative deviation. Line 9 then counts all feasible (w.r.t. that constraint alone) variable assignments through this arc that feature a given deviation from the right (recall that $\#_p^\ell(i, j)$ represents all paths from the left with a cumulative deviation of $\Delta_p^\ell(i, j)$ *or less*). Finally Line 10 returns the proportion of solutions with $x_i = v$. Going back to Figure 3, the processing of the arc from the bottom left vertex by Algorithm 4 will traverse left and right lists $\langle(6, 4), (4, 3), (2, 1)\rangle$ and $\langle(1, 3), (3, 2), (5, 4), (7, 1)\rangle$ respectively. Suppose that $\delta_\mu(v) = 1$ and $\Delta^{\max} = 10$: it will identify feasible combinations $(6, 4)(1, 3)$, $(6, 4)(3, 2)$, $(4, 3)(5, 4)$, and $(2, 1)(7, 1)$, in that order, for a total of $4 \times 3 + 4 \times 2 + 3 \times 4 + 1 \times 1 = 33$ solutions.

The cost of replacing single labels by lists, both in time and space, is a multiplicative factor corresponding to the number of pairs in these lists. Since there is no point in storing amounts of cumulative deviation larger than what is allowed by the constraint and since individual deviations vary in integral steps (even though deviation values may be fractional), that number of pairs is at most $\Delta^{\max} + 1$. That factor will tend to grow linearly with n (since Δ corresponds to a sum over the variables).

Algorithm 3: Computing the left labels on vertices of the layered graph.

input: layered graph G, upper bound on deviation $\Delta^{\max}$

output: set of left labels $L_{i,j}$ for each vertex (i,j) of G

$L_{0,0} \leftarrow \langle(0,1)\rangle$;
for $i = 1,2,\ldots,n$ **do**
  forall the *vertices* (i,j) *in layer* i *of* G **do**
    $L \leftarrow \langle\rangle$;
    forall the *arcs from a vertex* $(i-1,k)$ *to vertex* (i,j) **do**
      $p \leftarrow 1$;
      while $(p \le m^{\ell}_{i-1,k}) \wedge (\Delta^{\ell}_p(i-1,k) + \delta_\mu(j-k+\lfloor\mu\rfloor) > \Delta^{\max})$ **do**
        $p{+}{+}$;
      $q \leftarrow 1$;
      while $(q \le m^{\ell}_{i,j}) \wedge (p \le m^{\ell}_{i-1,k})$ **do**
        if $\Delta^{\ell}_p(i-1,k) + \delta_\mu(j-k+\lfloor\mu\rfloor) = \Delta^{\ell}_q(i,j)$ **then**
          insert $(\Delta^{\ell}_q(i,j), \#^{\ell}_q(i,j) + \#^{\ell}_p(i-1,k))$ at the back of L;
          $p{+}{+}$;
          $q{+}{+}$;
        else if $\Delta^{\ell}_p(i-1,k) + \delta_\mu(j-k+\lfloor\mu\rfloor) > \Delta^{\ell}_q(i,j)$ **then**
          insert $(\Delta^{\ell}_p(i-1,k) + \delta_\mu(j-k+\lfloor\mu\rfloor), \#^{\ell}_p(i-1,k))$ at the back of L;
          $p{+}{+}$;
        else
          insert $(\Delta^{\ell}_q(i,j), \#^{\ell}_q(i,j))$ at the back of L;
          $q{+}{+}$;
      if $q \le m^{\ell}_{i,j}$ **then**
        append $\langle(\Delta^{\ell}_q(i,j), \#^{\ell}_q(i,j)), \ldots, (\Delta^{\ell}_{m^{\ell}_{i,j}}(i,j), \#^{\ell}_{m^{\ell}_{i,j}}(i,j))\rangle$ to L;
      while $p \le m^{\ell}_{i-1,k}$ **do**
        insert $(\Delta^{\ell}_p(i-1,k) + \delta_\mu(j-k+\lfloor\mu\rfloor), \#^{\ell}_p(i-1,k))$ at the back of L;
    $t \leftarrow 0$;
    $L_{i,j} \leftarrow \langle\rangle$;
    forall the *elements* $(\Delta^{\ell}(i,j), \#^{\ell}(i,j))$ *of* L *traversed in reverse order* **do**
      $t \leftarrow t + \#^{\ell}(i,j)$;
      insert $(\Delta^{\ell}(i,j), t)$ at the front of $L_{i,j}$;

Algorithm 4: Solution density algorithm for the `dispersion` constraint

input: variable index i, value v

count $\leftarrow 0$;
forall the *arcs between a vertex* (i,j) *and vertex* $(i+1, j+(v-\lfloor\mu\rfloor))$ **do**
  $p \leftarrow 1$;
  for $q = 1, 2, \ldots, m^r_{i+1,j+(v-\lfloor\mu\rfloor)}$ **do**
    while $(p \leq m^l_{ij}) \wedge (\Delta^\ell_p(i,j) + \delta_\mu(v) + \Delta^r_q(i+1, j+(v-\lfloor\mu\rfloor)) > \Delta^{\max})$ **do**
      $p{+}{+}$;
    if $p > m^l_{ij}$ **then**
      break;
    count $\leftarrow$ count + $\#^\ell_p(i,j) \times \#^r_q(i+1, j+(v-\lfloor\mu\rfloor))$;
return count / $\#^\ell_1(n, n(\mu-\lfloor\mu\rfloor))$;

THEOREM 4. *Left and right lists of labels for layered graph G can be computed in $\mathcal{O}(n^2 d[D]\Delta^{\max})$ time.*

Proof As in Algorithms 1 and 2, the outer loops of Algorithm 3 visit each arc of the graph once, but here the appropriate lists of labels must then be traversed. □

THEOREM 5. *Given labeled layered graph G, Algorithm 4 computes the solution density of variable-value pair (x_i, v) in $\mathcal{O}(n[D]\Delta^{\max})$ time for the* `dispersion` *constraint with fixed mean.*

Proof The number of steps to process a single arc in Algorithm 4 is in $\mathcal{O}(\Delta^{\max})$ and the number of relevant arcs for a given variable-value pair (x_i, v) is bounded above by the number of vertices in layer i, which is in $\mathcal{O}(n[D])$ as we already saw in the proof of Theorem 2. □

4. Empirical Evaluation

In this section we evaluate both the filtering and the search guiding capabilities of our proposal. All experiments were run on Dual core AMD 2.1 GHz computers with 8 GB of memory, running Linux FC13 64 bits and using IBM ILOG Solver 6.7 and OscaR 1.0 (on top of Scala 2.10) (OscaR Team 2012).

We evaluate the impact of using our domain-consistency algorithm for `dispersion` instead of the bounds($\mathbb{Z}$)-consistency algorithms previously proposed by solving instances

of two benchmark problems from the literature and an extensive set of randomly generated instances for a synthetic problem in which the `dispersion` constraint is prominent. We also compare a branching heuristic based on solution densities, maxSD, to other generic branching heuristics as well as to some tailored heuristics. For each variable-value pair maxSD collects solution densities from the problem constraints offering such information and in which that variable appears, labels that pair with the maximum of the collection, and branches on the variable-value pair with the largest label (Zanarini and Pesant 2009). In our experiments only the `dispersion` constraint will provide solution densities (the other global constraints featured in our models are `gcc` and `binpacking` but currently we do not have implementations of solution density computation for them) so we simply branch on the variable-value pair with the highest solution density according to that constraint. We focus on measuring the size of the search tree but also address running time.

4.1. Synthetic Instances

For these experiments, we generated instances by starting with identical domains for variables and then randomly removing between a third and a half of the values in each domain. We varied both the number of variables (10, 20, 40) and the original domain size (10, 20, 30), and generated 50 instances for each of the nine combinations. The CP model consists of a `dispersion` constraint on the n variables with $\mu = \lfloor \frac{D^{\max}+D^{\min}}{2} \rfloor$ (an integer value in the middle of the original domain of the variables), and of a `gcc` constraint (global cardinality constraint (Régin 1996)) on the variables requiring that any value appear at most $\frac{n}{5}$ times. In order to challenge our filtering algorithms and branching heuristics we aimed for a mix of satisfiable and unsatisfiable instances through a careful choice of the deviation upper bound following some initial experiments: we set $\Delta \leq 2n$ for the L_2-deviation and $\Delta \leq 1.2n$ for the L_1-deviation. We use a 30-minute time limit.

Table 2 reports the number of instances solved in each set using L_1-deviation for the `dispersion` constraint and several combinations of consistency level and branching heuristic. The first two columns give the number of variables and the size of the original domains, respectively. The next seven columns give the number of instances solved using each combination, broken down into satisfiable and unsatisfiable, the last number representing unsolved instances. "DC" indicates our domain consistency algorithm for `dispersion` whereas "BC" refers to the light-weight linear-time bounds consistency algorithm described

Table 2 Number of synthetic instances solved (L_1-deviation). Entry "$a/b/c$" means that a instances were shown to be satisfiable, b instances were shown to be unsatisfiable, and c instances could not be solved within the 30-minute time limit.

n	[D]	BC lex/lex	DC lex/lex	BC dom/lex	DC dom/lex	BC dom/μ	DC dom/μ	DC maxSD
10	10	49/1/0	49/1/0	49/1/0	49/1/0	49/1/0	49/1/0	49/1/0
	20	46/4/0	46/4/0	46/4/0	46/4/0	46/4/0	46/4/0	46/4/0
	30	48/2/0	48/2/0	48/2/0	48/2/0	48/2/0	48/2/0	48/2/0
20	10	46/0/4	44/0/6	49/0/1	46/0/4	50/0/0	50/0/0	50/0/0
	20	44/1/5	40/1/9	45/1/4	41/1/8	48/1/1	48/1/1	48/1/1
	30	41/3/6	38/2/10	42/3/5	41/3/6	45/3/2	45/3/2	45/4/1
40	10					45/0/5	46/0/4	47/0/3
	20					27/0/23	31/0/19	38/0/12
	30					29/0/21	34/0/16	47/0/3

Table 3 Number of synthetic instances solved (L_2-deviation). Entry "$a/b/c$" means that a instances were shown to be satisfiable, b instances were shown to be unsatisfiable, and c instances could not be solved within the 30-minute time limit.

n	[D]	BC dom/lex	DC dom/lex	DC dom/μ	DC maxSD
10	10	49/1/0	49/1/0	49/1/0	49/1/0
	20	46/4/0	46/4/0	46/4/0	46/4/0
	30	48/2/0	48/2/0	48/2/0	48/2/0
20	10	46/0/4	50/0/0	50/0/0	50/0/0
	20	44/2/4	48/2/0	48/2/0	48/2/0
	30	41/4/5	45/5/0	45/5/0	45/4/1
40	10			49/0/1	48/0/2
	20			38/0/12	39/0/11
	30			43/0/7	46/0/4

in Schaus et al. (2007b) which we implemented in IBM ILOG Solver (that algorithm requires μ to be an integer, which is why we used an integral mean for our synthetic instances). As for branching heuristics: "lex/lex" stands for lexicographic variable and value selection, a static heuristic; "dom/lex" stands for smallest-domain-first variable selection and lexicographic value selection; "dom/μ" is as the previous one except that values are selected from the mean outwards, a sensible heuristic given the upper bound on global deviation from the mean; and "maxSD" applies the maxSD generic branching heuristic. Note that there is no "BC maxSD" combination since the computation of solution densities requires the data structures put in place to enforce domain consistency so we already pay the price for that stronger inference. Every other aspect is identical in each variant and since IBM ILOG Solver was used throughout we know precisely what is being measured.

Table 3 reports the number of instances solved in each set using L_2-deviation for the `dispersion` constraint but on fewer combinations. Here a comparison is more delicate because the DC implementation is written in IBM ILOG Solver whereas we use the BC

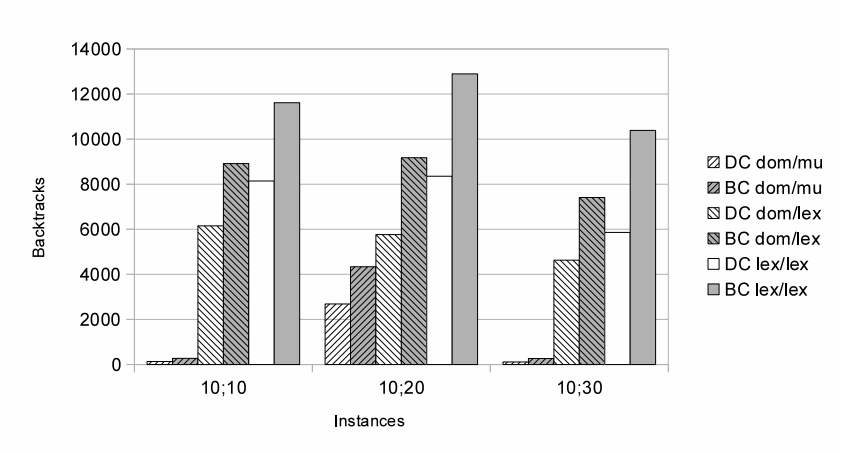

Figure 4 **Total number of backtracks on all instances of a set using L_1-deviation. The difference in search effort between bounds and domain consistency is shown for three branching heuristics.**

implementation of L_2-deviation available in OscaR through their `spread` constraint and therefore run our experiments for that variant on a different solver.

Inference. Looking first at Table 2 and comparing pairs of BC/DC variants allows us to measure the impact of the level of consistency being enforced. Every 10-variable instance is solved within the time limit by each combination. Figure 4 reports the total number of backtracks required in each case. The lex/lex combinations provide us with a static branching heuristic that will therefore not be influenced by the level of inference being applied: we note that search effort is increased by about 50% if we apply bounds consistency instead of domain consistency. That percentage remains about the same for the other two dynamic branching heuristics. Although not shown in the figure, a comparison of the dom/μ variants on the 20-variable instances, for which both variants solved exactly the same instances, reveals that search effort on satisfiable instances grows to about an 8-fold increase.

On 20-variable instances we note that for the less successful lex/lex and dom/lex heuristics, bounds consistency solves more instances than domain consistency. This is probably because the bounds consistency algorithm, specialized for L_1-deviation and an integral mean, is very fast: we estimate that it leads to about seven times more backtracks per second on the 20;10 instances whereas the increase in search effort is about 3-fold for lex/lex. The 40-variable instances were considered out of reach for the latter heuristics but for the dom/μ variants we observe that BC starts solving fewer instances than DC.

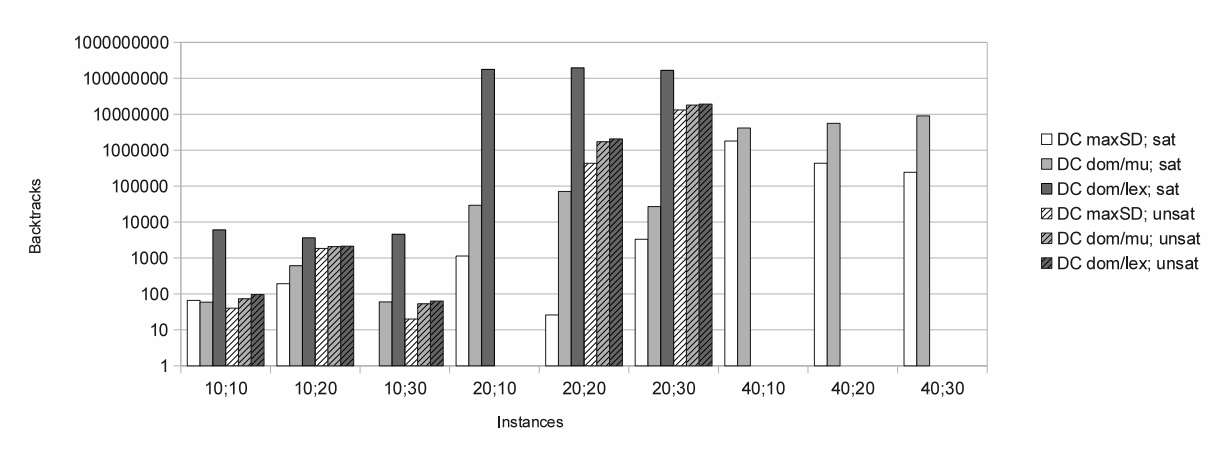

Figure 5 **Total number of backtracks on solved instances of a set using L_1-deviation. Three branching heuristics are evaluated with our domain consistency algorithm. Satisfiable and unsatisfiable instances are reported on separately.**

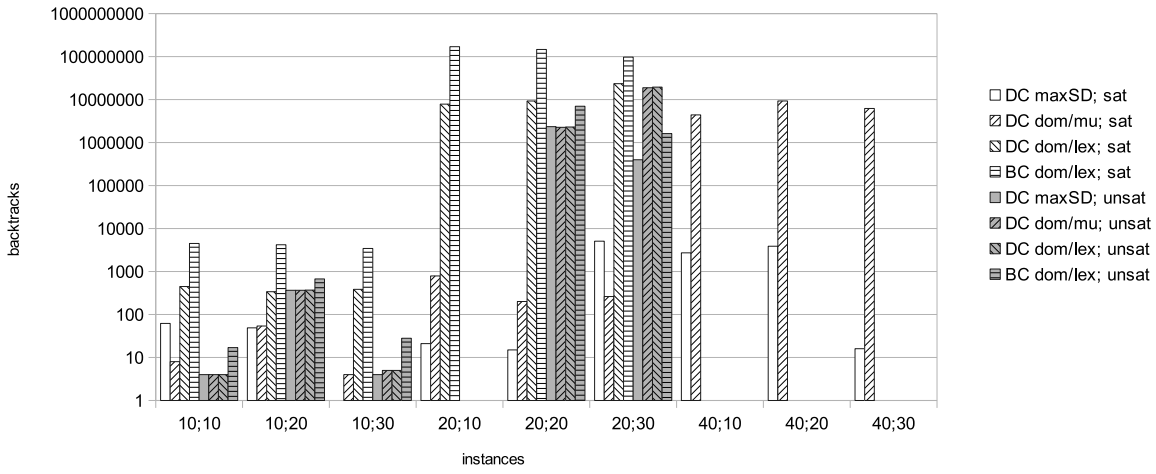

Figure 6 **Total number of backtracks on solved instances of a set using L_2-deviation. Four combinations of consistency level and branching heuristic are evaluated. Satisfiable and unsatisfiable instances are reported on separately.**

Turning to Table 3, comparing the two variants of dom/lex reveals that starting with 20-variable instances, DC solves more instances (keep in mind though that different solvers are used). However it also requires about one order of magnitude fewer backtracks on satisfiable instances and does almost as well on unsatisfiable instances (see Figure 6). Here too the 40-variable instances were out of reach.

Search. For L_1-deviation and the three dynamic branching heuristics with DC (see Table 2 and Figure 5), dom/lex trails the other two from the start whereas dom/μ, a problem-

specific heuristic, starts falling behind generic heuristic maxSD in number of backtracks (20-variable instances) and ultimately in number of instances solved (40-variable instances). Note that maxSD is backtrack-free on every satisfiable 10;30 instance and across all satisfiable instances it backtracks on at most a few instances in each set of 50.

For L_2-deviation the DC combinations, all enforcing domain consistency and implemented in IBM ILOG Solver, provide a comparison of branching heuristics (see Table 3 and Figure 6). The number of 10- and 20-variable instances solved by each is practically the same but there is a marked difference between the number of backtracks for dom/lex and for the other two on satisfiable instances. The overall results are mixed between generic heuristic maxSD and problem-specific heuristic dom/μ for the smaller instances but the former dominates on the larger 40-variable instances. Again maxSD is backtrack-free on every satisfiable 10;30 instance and seldom backtracks on satisfiable instances.

As an indication of the increase in computational effort when using heuristic maxSD as opposed to a much simpler one such as dom/μ, we observed that search with the latter performed about 1.5 times more backtracks per second, which appears reasonable given that about the same factor is observed between dom/μ and static heuristic lex/lex.

4.2. Nurse to Patient Assignment Problem (NPAP)

This application of L_2-deviation is set in a neonatal intensive care unit and asks for a balanced workload for nurses being assigned newborn patients requiring various amounts of care (*acuity*). A nurse's workload is the sum of the acuities of the patients assigned to him. Patients each belong to a zone — a nurse can only work in one zone and there is an upper limit both on the number of patients in his care and on the corresponding workload (these limits are respectively 3 and 105 for the benchmark instances considered). It was introduced in Mullinax and Lawley (2002) but the Mixed Integer Programming approach used was not satisfactory. Schaus et al. (2009) proposed a CP approach using the `spread` constraint that pre-computes the number of nurses for each zone and then solves each zone separately. In a follow-up paper they show that this decomposition strategy produces provably optimal solutions on all but two instances considered (Schaus and Régin 2013).

The instances we used for benchmarking were randomly generated in Schaus and Régin (2013) using a realistic statistical model proposed in Mullinax and Lawley (2002). They range from 3 to 20 zones and up to 102 nurses and 258 patients.

The CP model for this problem uses one variable n_i per patient i indicating which nurse it is assigned to and one variable w_j per nurse j indicating his workload. There is also another variable d representing the L_2-deviation from the average of the w_j values, to be minimized. On these variables are expressed a `dispersion` constraint to link d to the w_j variables, a `gcc` constraint on the n_i variables to limit the number of patients for each nurse, and a `binpacking` constraint (Shaw 2004) linking the n_i and w_j variables while considering patient acuities. We refer the reader to Schaus and Régin (2013) for additional details on the problem, the benchmark instances, or the model.

To this model Schaus and Régin (2013) add a branching heuristic that chooses the next n_i variable to branch on according to the smallest-domain-first selection criterion and the next value (nurse) to assign to it by considering nurses already assigned a patient plus one yet-unused nurse, in lexicographic order. This value selection criterion has the advantage of breaking symmetries among nurses dynamically during search. We will refer to this heuristic as "SR13".

We propose a simple static branching heuristic focusing on the load variables: we first select the w_j variables in lexicographic order and then the n_i variables in decreasing order of acuity; values are selected in lexicographic order. We break symmetries among nurses statically by adding constraints $w_j \geq w_{j+1}$ to the model. We also initially bound deviation d from above using a greedy solution obtained by considering patients in decreasing order of acuity and assigning them to the least loaded available nurse. We will refer to our heuristic as "static". As before we refer to implementations enforcing domain consistency on `dispersion` as "DC" and to others enforcing bounds consistency as "BC".

Table 4 reports the number of backtracks on the benchmark instances for the different implementations. Each line corresponds to an individual instance and the first three columns give the number of zones, nurses, and patients for that instance.

Looking first at how well the problem is being solved, Column BC SR13 corresponds to the implementation of Schaus and Régin (2013) whereas DC static is what we proposed: our search trees are smaller than theirs, often significantly so. To be fair we also ran SR13 with the same upper bound on deviation we used (denoted BC SR13*): as expected this reduces the number of backtracks but not enough to be competitive with DC static except for the 15-zone instance (for DC static the number of backtracks is dominated by one bad zone). Column DC maxSD corresponds to first branching on the workload variables, as

Table 4 Number of backtracks during search for the six variants considered to solve NPAP benchmark instances from Schaus and Régin (2013).

zones	nurses	patients	DC static	BC static	BC SR13	BC SR13⋆	DC SR13⋆	DC maxSD
3	15	42	147	11296	11973	4424	2029	121
3	18	43	939	30541	4473	1129	3262	563
3	17	43	4211	497979	17155	13289	16254	2964
3	17	42	142	20446	14089	4526	896	168
3	18	43	2110	1802706	43421	20157	158588	2049
3	14	38	198	13926	3335	2750	1504	180
3	19	48	2819	730997	23041	13065	5868	3132
3	16	44	328	32571	19817	8240	7955	234
3	19	49	1927	573272	20370	16886	20115	2824
3	17	41	157	60393	7606	1744	775	175
6	31	78	1387	84347	12019	7569	3364	1210
15	71	198	33269	93641	38651	10353	96411	33319
20	102	258	28807	1659405	1176852	1082651	149538	28233

in DC static, but using the maxSD heuristic (handling ties by favoring a value closest to the mean) and then proceeding as in DC static i.e. branching on the nurse assignment variables by decreasing acuity and selecting nurses in lexicographic order: its performance is comparable to that of DC static, neither one dominating the other. As mentioned before, comparing running times is more delicate since the DC variants are written in ILOG-IBM Solver whereas the BC ones are written in OscaR. As one indication, DC static solves the largest instance (20 zones) in 2.73 seconds on a 2,1 GHz machine whereas Schaus and Régin (2013) report 25.17 seconds for BC SR13 on the same instance (but they do not specify the computer used).

Turning now to branching heuristics, a good opportunity to compare static and SR13 is given by columns DC static and DC SR13⋆ since both are run on the same solver using exactly the same model, upper bound on deviation, and consistency level for `dispersion`: the former is clearly more effective though the results in Column BC static suggest that static requires domain consistency in order to perform well.

Finally to assess the advantage of domain consistency for this problem, pairs DC static / BC static and DC SR13⋆ / BC SR13⋆ provide some comparison for two instances of branching heuristics: DC always performs better in the first case and 8 times out of 13 in the second case.

So in summary we improve the state of the art on this problem, proposing a heuristic that branches on load variables first and that works very well in combination with domain consistency. A variant of that branching heuristic that exploits solution densities on load variables performs well but not consistently better than our tailored heuristic.

4.3. Balanced Academic Curriculum Problem (BACP)

Recall that the BACP asks to assign courses to semesters so as to balance the academic load between semesters, defined as the sum of the number of credits from courses assigned to a semester. In addition there are a minimum and maximum number of courses per semester and some courses are prerequisite to others, e.g. course B must be assigned to a semester occurring after the one to which course A is assigned. In the literature the L_2-deviation is typically used to measure balance.

From the largest instance in Gent and Walsh (1999) 100 instances were generated by randomly assigning between 1 and 5 credits to courses and by randomly choosing a subset of the prerequisites (Schaus 2009).[1] They feature 66 courses, 12 semesters, 50 prerequisite pairs, and use 5 and 7 as the minimum and maximum number of courses per semester respectively.

The CP model for this problem uses one variable s_i per course i indicating which semester it is assigned to and one variable ℓ_j per semester j indicating its academic load. There is also another variable d representing the L_2-deviation from the average of the ℓ_j values, to be minimized. On these variables are expressed a `dispersion` constraint to link d to the ℓ_j variables, a `gcc` constraint on the s_i variables to limit the number of courses for each semester, a `binpacking` constraint linking the s_i and ℓ_j variables while considering course credits, and a binary inequality constraint between a pair of s_i variables for each prerequisite relation.

As branching heuristic we choose the next s_i variable to branch on according to the smallest-domain-first selection criterion and the next value (semester) to assign to it by favoring the semester currently with the smallest academic load, as originally proposed in Schaus (2009).

The instances solved in Schaus (2009) and Monette et al. (2013) do not include the original restriction on the number of courses per semester — we take it into account here, which makes the instances significantly harder to solve to optimality.

Figure 7 reports the percentage of instances that could be solved to optimality within a given number of backtracks. We set a 20-minute time limit on individual runs. Solid curves BC dom;least_loaded and DC dom;least_loaded respectively report the performance of

[1] available from `http://becool.info.ucl.ac.be/resources/bacp`

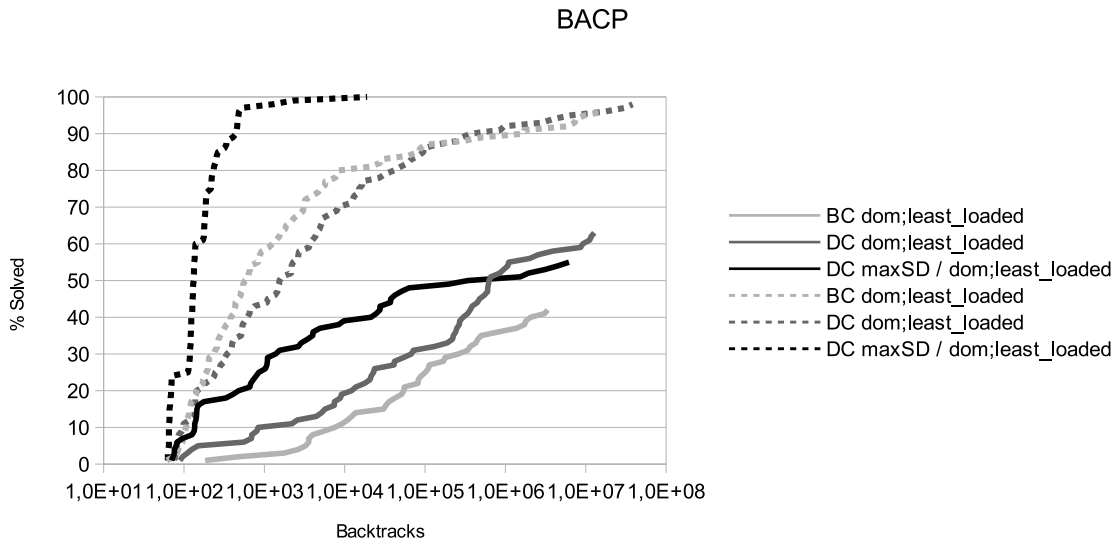

Figure 7 **Percentage of instances solved with respect to the number of backtracks during search for the three variants considered on BACP benchmark instances from Schaus (2009). Solid curves correspond to the full problem whereas dashed curves solve a relaxed version without the restriction on the number of courses per semester.**

the branching heuristic previously described using the bounds-consistency or the domain-consistency algorithms for the `dispersion` constraint: the latter curve lies above the former and for a given percentage of instances solved, its required number of backtracks is up to one order of magnitude smaller. We observe that a significant percentage of the instances cannot be solved within the time limit, which confirms that these instances are challenging: as a baseline, the smallest-domain-first;lexicographic variable-value branching heuristic (not shown on the figure) only manages to solve 3% of the instances. The looser instances (i.e. without the stated restriction on the number of courses per semester) previously solved in the literature are significantly easier: as an indication, dashed curve DC dom;least_loaded shows the performance of the domain-consistency version if we allow any number of courses per semester, solving 98% of the instances within the same time limit.

We also observe that the maximum number of backtracks reached within a fixed time limit is not so different between the curves, which again shows that the domain-consistency algorithm does not present a significant computational overhead in practice.

We next consider branching heuristic maxSD / dom;least_loaded which first branches on the load variables using the maxSD heuristic (and handling ties by favoring a value closest to the mean) and then branches on the semester assignment variables according to

the dom;least_loaded heuristic. For the looser instances the dashed curves show the clear superiority of DC maxSD / dom;least_loaded over DC dom;least_loaded: all 100 instances are solved to optimality in less than a second each, almost all of them within a few hundred backtracks. In comparison Monette et al. (2013), implementing BC dom;least_loaded, report that 33 out of 100 instances required more that a second to be solved to optimality and even that two instances timed out after 12 hours.

On the original instances (with the restriction on the number of courses per semester) solid curves DC maxSD / dom;least_loaded and DC dom;least_loaded show that, up to a few hundred thousand backtracks, significantly more instances are solved by first branching on the load variables according to solution densities. Beyond that limit the relative performance is reversed, albeit to a lesser degree. Given the ease with which DC maxSD / dom;least_loaded solved the looser instances, that behavior is likely due to the tight restriction on the number of courses — load variables are branched on first, sometimes fixing the number of credits in each semester into a configuration that may not be feasible and whose refutation tree may be large.

We also noticed that for an optimization problem such as this in which the deviation from a fixed mean is sought to be minimized, maxSD is helped by the fact that the initial domains of the load variables were defined with that mean at their centre: otherwise the load values initially recommended by the heuristic as most frequent in solutions to the `dispersion` constraint may not coincide at all with values close to the mean (recall Example 3).

In summary maintaining domain consistency and branching according to solution densities each improve our ability to solve this problem.

5. Conclusion

In this paper we considered constraints to express balance, important in many combinatorial problems, and presented new algorithms to eliminate inconsistent variable assignments and to count the number of solutions. We provided empirical evidence that these can lead to significant practical improvements in combinatorial problem solving. In particular we improved the state of the art on the Nurse to Patient Assignment Problem and the Balanced Academic Curriculum Problem.

The contributions of this paper with respect to `dispersion` constraints are: the first practical algorithm that achieves domain consistency, an approach that equally applies

to `spread` and `deviation` constraints or other metrics, the ability to filter not only from the maximum allowed deviation (as previous contributions did) but alternately from the minimum allowed deviation, and the possibility of counting the number of solutions exactly, which can be used in counting-based search heuristics.

Being able to express balance declaratively at the modeling level is very convenient for non-expert users as it avoids the need to handle it operationally within a hand-crafted search procedure. The present contribution does not introduce this modeling ability but makes it more powerful by improving its filtering capability. Arguably the main obstacle of constraint programming technology for non-experts is the need to write a dedicated search heuristic: existing generic search heuristics often aren't robust enough to solve industrial problems reliably. Competing technologies such as SAT and MIP solvers typically do not require users to tailor search. Counting-based search heuristics are generic and have shown very promising robustness. Our contribution with respect to these heuristics is to provide the required counting information for `dispersion` constraints, thereby broadening their applicability to solve combinatorial problems.

References

Demassey, S., G. Pesant, L.-M. Rousseau. 2006. A Cost-Regular Based Hybrid Column Generation Approach. *Constraints* **11** 315–333.

Falkenauer, E. 2005. Line Balancing in the Real World. *Proc. Int. Conf. Product Lifecycle Management*. Inderscience Enterprises Ltd, 360–370.

Gent, I.P., T. Walsh. 1999. CSPLib: a benchmark library for constraints. Tech. rep., APES-09-1999. Problem library available at http://www.csplib.org. A shorter version of the paper appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).

Lemaître, M., G. Verfaillie, N. Bataille. 1999. Exploiting a Common Property Resource under a Fairness Constraint: a Case Study. T. Dean, ed., *IJCAI*. Morgan Kaufmann, 206–211.

Monette, J.-N., N. Beldiceanu, P. Flener, J. Pearson. 2013. A Parametric Propagator for Discretely Convex Pairs of Sum Constraints. C. Schulte, ed., *CP*, *Lecture Notes in Computer Science*, vol. 8124. Springer, 529–544.

Mullinax, C, M Lawley. 2002. Assigning patients to nurses in neonatal intensive care. *J Oper Res Soc* **53** 25–35. URL `http://dx.doi.org/10.1057/palgrave.jors.2601265`.

OscaR Team. 2012. OscaR: Scala in OR. Available from `https://bitbucket.org/oscarlib/oscar`.

Pesant, G. 2008. Constraint-Based Rostering. *Proc. 7th Int. Conf. Practice and Theory of Automated Timetabling (PATAT)*. 11 p. Available at http://www.asap.cs.nott.ac.uk/external/watt/patat/patat08/Papers/Pesant-WB1.pdf.

Pesant, G., J.-C. Régin. 2005. sPREAD: A Balancing Constraint Based on Statistics. P. van Beek, ed., *CP*, *Lecture Notes in Computer Science*, vol. 3709. Springer, 460–474.

Régin, J.-C. 1996. Generalized Arc Consistency for Global Cardinality Constraint. *Proceedings of the Thirteenth National/Eighth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, AAAI-98/IAAI-98*, vol. 1. 209–215.

Schaus, P. 2009. Solving Balancing and Bin-Packing problems with Constraint Programming. Ph.D. thesis, Université catholique de Louvain.

Schaus, P., Y. Deville, P. Dupont. 2007a. Bound-Consistent Deviation Constraint. C. Bessiere, ed., *CP*, *Lecture Notes in Computer Science*, vol. 4741. Springer, 620–634.

Schaus, P., Y. Deville, P. Dupont, J.-C. Régin. 2007b. The Deviation Constraint. P. Van Hentenryck, L. A. Wolsey, eds., *CPAIOR*, *Lecture Notes in Computer Science*, vol. 4510. Springer, 260–274.

Schaus, P., P. Van Hentenryck, J.-C. Régin. 2009. Scalable Load Balancing in Nurse to Patient Assignment Problems. W. J. van Hoeve, J. N. Hooker, eds., *CPAIOR*, *Lecture Notes in Computer Science*, vol. 5547. Springer, 248–262.

Schaus, Pierre, Jean-Charles Régin. 2013. Bound-Consistent Spread Constraint. *EURO Journal on Computational Optimization* 1–24doi:10.1007/s13675-013-0018-8. URL `http://dx.doi.org/10.1007/s13675-013-0018-8`.

Shaw, Paul. 2004. A Constraint for Bin Packing. Mark Wallace, ed., *CP*, *Lecture Notes in Computer Science*, vol. 3258. Springer, 648–662.

Trick, M. A. 2003. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Annals of Operations Research* **118** 73–84.

Zanarini, A., G. Pesant. 2009. Solution Counting Algorithms for Constraint-Centered Search Heuristics. *Constraints* **14** 392–413.