

Explaining Propagation for Gini and Spread with Variable Mean

Alexander Ek  



Dept. of Data Science & AI, Monash University, Melbourne, Australia
CSIRO Data61, Melbourne, Australia

Andreas Schutt  

CSIRO Data61, Melbourne, Australia

Peter J. Stuckey  

Dept. of Data Science & AI, Monash University, Melbourne, Australia

Guido Tack  

Dept. of Data Science & AI, Monash University, Melbourne, Australia

Abstract

In optimisation problems involving multiple agents (stakeholders) we often want to make sure that the solution is balanced and fair. That is, we want to maximise total utility subject to an upper bound on the statistical dispersion (e.g., spread or the Gini coefficient) of the utility given to different agents, or minimise dispersion subject to some lower bounds on utility. These needs arise in, for example, balancing tardiness in scheduling, unwanted shifts in rostering, and desired resources in resource allocation, or minimising deviation from a baseline in schedule repair, to name a few. These problems are often quite challenging. To solve them efficiently we want to effectively reason about dispersion. Previous work has studied the case where the mean is fixed, but this may not be possible for many problems, e.g., scheduling where total utility depends on the final schedule. In this paper we introduce two log-linear-time dispersion propagators—(a) spread (variance, and indirectly standard deviation) and (b) the Gini coefficient—capable of explaining their propagations, thus allowing effective clause learning solvers to be applied to these problems. Propagators for (a) exist in the literature but do not explain themselves, while propagators for (b) have not been previously studied. We avoid introducing floating-point variables, which are usually not supported by learning solvers, by reasoning about scaled, integer versions of the constraints. We show through experimentation that clause learning can substantially improve the solving of problems where we want to bound dispersion and optimise total utility and vice versa.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence; Theory of computation → Constraint and logic programming

Keywords and phrases Spread constraint, Gini index, Filtering algorithm, Constraint programming, Lazy clause generation

Digital Object Identifier 10.4230/LIPIcs.CP.2022.7

Supplementary Material *Software*: www.github.com/aekh/chuffed

Dataset (Instances, Models, and Results): www.github.com/aekh/CP22-Extras

Funding This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Project ID IC200100009.

1 Introduction

In many real-world applications of combinatorial optimisation the statistical dispersion (sometimes called variability or spread) of the variable assignments in the solutions are important to consider. This is because some kind of balance or fairness within the solution



© Alexander Ek, Andreas Schutt, Peter J. Stuckey, and Guido Tack;
licensed under Creative Commons License CC-BY 4.0

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).

Editor: Christine Solnon; Article No. 7; pp. 7:1–7:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is desired. As an example, consider a resource allocation problem, where a set A of agents compete over a set R of resources under complex constraints. The utility that an agent $a \in A$ receives from being allocated a resource $r \in R$ may completely depend on what other resources a are also being allocated. For example, a woodworker gets little utility from being allocated nails but not a hammer, or hammer and nails but not planks. In this situation, it may be essential to balance the utilities (or metrics derived from the utilities) each agent receives. A fixed mean is often assumed in the literature, simplifying the constraint; however, fixing the mean is not possible in many contexts. Throughout this paper, we assume the mean is variable.

The **main contribution** in this paper is the development of efficient and explaining propagators, filtering the lower bound, for two kinds of dispersion measured over an array of variables X : (i) spread (i.e., variance and standard deviation), explained in Section 3; and (ii) the Gini coefficient, explained in Section 4. We only consider filtering the lower bound because usually minimisation of dispersion or keeping dispersion under some upper bound is of interest. Problems which require maximising dispersion or ensuring sufficient dispersion need filtering algorithms for the upper bound, and require other types of approximations and roundings than presented here, which we leave to future work.

A dispersion propagator can also be devised that propagates the bounds of the variables X whose dispersion is being measured. Interestingly we performed some preliminary experiments, where we implemented a naive domain consistent propagator for spread by checking each possible assignment to the variables X , and removing unsupported values. This propagator, which is the strongest possible, rarely removed values from the X variables before all but one of them were fixed. This is because the X variables act in aggregate in a large sum, and propagation is notoriously weak for reasoning about large sums. This is even more pronounced in our case because of the squared differences in the sum for variance and absolute differences in the sum for Gini. Hence, we decided not to pursue algorithms for this case, since it was clear they would rarely help.

In Sections 3 and 4 we respectively present lower bounds on spread and the Gini coefficient (Gini) and how to explain the propagation of those. We avoid introducing floating-point variables, which are usually not supported by clause learning solvers, by reasoning over appropriately scaled integer versions of the constraints. Selecting a scale is application-specific: there is a trade-off between the size of integers and fidelity.

2 Background

We briefly introduce main concepts used in this paper starting with those ones in combinatorial optimisation, statistics, and ending with real relaxations.

2.1 Combinatorial Optimisation

A constrained optimisation problem (COP) $P = (X, D, C, o)$ consists of a set of variables X , an initial domain D mapping each $x \in X$ to a set of integer values $x \mapsto D(x) \subseteq \mathbb{Z}$, a set of constraints C over the variables X and an objective function o to be minimised (w.l.o.g). An assignment θ is a mapping from each variable $x \in X$ to a value from its domain $\theta(x) \in D(x)$. An assignment θ is a *solution* of COP P if it makes all constraints $c \in C$ true. An assignment θ is an *optimal solution* if $\theta(o) \leq \theta'(o)$ for all solutions θ' of P .

An often effective way to solve COPs is the *constraint programming* (CP) solving technology [14]. In CP, each constraint $c \in C$ has a *propagator* f_c , which uses specialised algorithms and logic to reduce the domains of the variables concerning c when invoked, i.e.,

$f_c(D)(x) \subseteq D(x)$. Only guaranteed non-solutions are removed. If $f_c(D)(x) = \emptyset$ for some $x \in X$, we say that we have reached a failure because there is no solution in D . When no more inference can be made by the propagators, and we do not have a solution or failure yet, we need to search for a solution. This can be done by arbitrarily reducing the domain of a variable $x \in X$ to $D'(x) \subsetneq D(x)$, creating a binary branch point in the search tree of the constraints $x \in D'(x)$ in one child and $x \in D(x) \setminus D'(x)$ in the other.

We use $l..u$ to denote the set of integers $\{l, l+1, \dots, u-1, u\}$, and for a decision variable x with domain $D(x)$, we use \bar{x} to denote $\max(D(x))$ and \underline{x} to denote $\min(D(x))$.

An often effective augmentation of CP is lazy clause generation (LCG), which combines the techniques of Boolean satisfiability solving (SAT) and CP [9]. An LCG solver connects Boolean variables concerning bounds (and fixed values, but they will not be needed here) to the integer variables. The Boolean $\llbracket x \geq d \rrbracket$ holds if the integer variable x takes a value greater than or equal to d . We use notation $\llbracket x < d \rrbracket$ for $\neg \llbracket x \geq d \rrbracket$ and $\llbracket x \leq d \rrbracket$ for $\neg \llbracket x \geq d-1 \rrbracket$. An LCG solver tracks the reasons justifying all propagated domain changes happening during search. We call such reasons for each propagation an *explanation*, represented as a set of Boolean clauses, using the literals defined above, which were implied by the domains at the time of propagation and imply the propagation. In case of a failure, these explanations are used to find *nogoods*, which extract the reason for the failure as a new (previously implicit) constraint, representing an erroneous choice made earlier during search and preventing it from reoccurring. We can then backjump to the point just before this choice was made and add the new constraint explicitly to prevent making the same erroneous choice again. A popular LCG solver is Chuffed [3], which we will use and extend in this paper.

2.2 Statistical Preliminaries

Perhaps the most common statistical summary of a set of numbers is that of its central tendency, and arguably the most prevalent of those is the *arithmetic mean*. Suppose we have a *population* X of *integer* values x_1, \dots, x_n . We respectively denote the *arithmetic mean* and the *sum* of these as:

$$\mu_X = \frac{1}{n} \sum_{i=1}^n x_i, \quad M_X = \sum_{i=1}^n x_i.$$

For brevity, we will omit the subscript “ X ” when there is little room for ambiguity.

A *measure of dispersion* is another kind of statistical summary of a set of numbers. It describes how different or similar values of X are. The first measure of dispersion we consider is the *population variance*, or simply *variance*. The *variance* of X is defined as the average squared difference from the arithmetic mean, with its conventional and alternative formulations as follows:

$$\sigma_X^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2, \quad \sigma_X^2 = \left(\frac{1}{n} \sum_{i=1}^n x_i^2 \right) - \mu^2. \quad (1)$$

A related measure of dispersion is the *standard deviation*, which is the square-root of variance:

$$\sigma_X = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2},$$

Standard deviation is a monotonic transformation of variance. As such, minimising (and maximising) standard deviation is equivalent to minimising (and maximising) variance. Similarly, any constraints specified on the standard deviation can be squared to obtain the equivalent constraint on the variance. As such we only need to define a propagator for variance.

7:4 Explaining Propagation for Gini and Spread with Variable Mean

The second measure of dispersion we consider is the *Gini coefficient*. The *Gini coefficient* of X is conventionally formulated as:

$$\text{Gini}(X) = \frac{\sum_{i=1}^n \sum_{j=i+1}^n |x_i - x_j|}{n \sum_{i=1}^n x_i}.$$

Calculating this naïvely takes $\mathcal{O}(|X|^2)$ time, but an alternative linear-time formulation is well-known if X is sorted [4]:

$$\text{Gini}(X) = \frac{\sum_{i=1}^n (2i - n - 1)x_i}{n \sum_{i=1}^n x_i}. \quad (2)$$

The Gini coefficient, originally developed for measuring income inequality, measures how far a distribution X deviates from a totally equal distribution. Note that it is only meant to apply when $x_i > 0$, since otherwise the sum divisor can be zero.

2.3 Real Relaxation

We will examine the case of the dispersion constraints on a set of integer valued variables X . It will often be useful to relax the integrality constraint in order to create lower bounds on dispersion values.

In our lower bound calculations we relax the problem from integer variables to real variables.

► **Definition 1.** An \mathbb{R} -assignment of variables X is a mapping θ from X to \mathbb{R} such that $\underline{x} \leq \theta(x) \leq \bar{x}, \forall x \in X$.

We further make use of ν -centred assignments as defined by [13].

► **Definition 2.** Given a real value ν , an \mathbb{R} -assignment θ is ν -centred if

$$\theta(x) = \begin{cases} \bar{x} & \text{if } \bar{x} \leq \nu \\ \underline{x} & \text{if } \underline{x} \geq \nu \\ \nu & \text{otherwise} \end{cases}$$

We denote the ν -centred assignment θ_ν .

3 The Spread Constraint

The spread constraint [13] as initially introduced was defined as $\text{spread}(X, \mu, \sigma, \tilde{x})$, where μ is constrained to be the mean of the array of (say n) variables $X = [x_1, \dots, x_n]$, σ the standard deviation, and \tilde{x} the median. We will examine a slightly different form.

Let $\text{spread}(X, M, n2V)$ be defined to constrain M to be the sum of X , i.e., $\sum_{i=1}^n x_i$, and $n2V$ to be equal to n^2 times the variance of X , i.e., $n^2\sigma^2$. The advantage of this form is that each of the variables takes integer values. By making use of the alternate expression of variance, the following equations hold:

$$n2V = n \sum_{i=1}^n x_i^2 - M^2 \quad \wedge \quad M = \sum_{i=1}^n x_i, \quad (3)$$

Since all the variables involved are integer, we can use this within a solver that does not support floating-point variables, without difficulty.

But when n is large, the size of the $n2V$ may often push the boundaries of integers that most solvers can deal with. Thus, we also introduce a relaxed version of the spread constraint that works with a given fixed positive scaling factor s defined as follows.

`spread`(X, M, v, s) constrains M to be the sum of X and $v = \lfloor \sigma^2 \cdot s \rfloor$, where σ^2 is the variance of X scaled (by s) and rounded down. For instance, $s = 1$ means whole number and $s = 100$ means percentage. Note when this is used as a lower bound it is always correct. The benefit of this relaxed form of the constraint is that it, again, only involves integer variables.

We can define this as a decomposition in MiniZinc using Equation (3), as follows:

```

1 predicate spread(array[int] of var int:X, var int:M, var int:v, int:s) =
2   let { int: n=length(X);
3     array[int] of var 0..infinity: sq = [x * x | x in X];
4     var 0..n*ub(sum(sq)): v_ = n*sum(sq);
5     var 0..max(lb(M)*lb(M), ub(M)*ub(M)): Msq = M*M;
6     var 0..ub(v_): v__ = v_ - Msq;
7     var 0..ub(v__)*s: v___ = v__*s; }
8   in v = v___ div (n*n) /\ M = sum(X);

```

Note that `lb` and `ub` return the best known, *at compile time*, lower and upper bound respectively of the argument expression. Note that the first version we defined is implemented by `spread(X, M, n2v) = spread(X, M, n2v, n2)`.

3.1 Lower Bound on Variance

In this section we present a log-linear-time propagator for filtering the lower bound of the variance variable using binary chop.

We define a real-value relaxed lower bound formula for the variance:

$$LBV(X, M) = \min_{m=M}^{\overline{M}} LBV'(X, \frac{m}{n}), \quad \text{where } LBV'(X, \nu) = \frac{1}{n} \sum_{i=1}^n (\theta_\nu(x_i) - \nu)^2$$

Essentially, we pick the ν -centred assignment with the lowest variance. From a previous result by [13] we know that a variance-minimal solution to the spread constraint must be a ν -centred assignment.

► **Lemma 3.** $LBV(X, M)$ is a lower bound on σ^2 , i.e., $LBV(X, M) \leq \sigma^2$.

Proof. Let $Y = \{y_1, \dots, y_n\}$ be an arbitrary assignment of the variables in X where $y_i \in D(x_i)$ and $\nu = M_Y/n$ be the mean of Y where $M_Y = \sum_{i=1}^n y_i$. Now, we only need to prove that $LBV'(X, \nu) \leq \frac{1}{n} \sum_{i=1}^n (y_i - \nu)^2$, i.e., the variance of the ν -centred assignment is less than or equal to the variance of the assignment Y . We do so by proving $(\theta_\nu(x_i) - \nu)^2 \leq (y_i - \nu)^2$ for every i . By construction of the ν -centred assignment (see Definition 2), for any i , where $1 \leq i \leq n$, it holds either that $y_i \leq \theta_\nu(x_i) \leq \nu$ for the case $\bar{x}_i \leq \nu$, that $\nu \leq \theta_\nu(x_i) \leq y_i$ for the case $\bar{x}_i \geq \nu$, or that $\nu = \theta_\nu(x_i)$ (i.e., $(\theta_\nu(x_i) - \nu)^2 = 0$ otherwise). Thus, it holds $(\theta_\nu(x_i) - \nu)^2 \leq (y_i - \nu)^2$ in all cases. Hence, the lemma holds. ◀

We can calculate LBV in log-linear time, because LBV' is convex in ν .

► **Lemma 4.** $LBV'(X, \nu)$ is a convex function in ν .

Proof. Since a sum of convex functions is convex, we only need to prove for each i , where $1 \leq i \leq n$, that the function $f(\nu) = (\theta_\nu(x_i) - \nu)^2$ is convex in ν . By construction of θ_ν (see Definition 2), the difference $\theta_\nu(x_i) - \nu$ is strictly decreasing until $\nu = \underline{x}_i$, then zero until $\nu = \bar{x}_i$, and strictly decreasing again. Since the difference is squared, the function f is a quadratic function with a flat bottom of some length. Hence, it is convex and $LBV'(X, \nu)$ as well. ◀

■ **Algorithm 1** Only run if not all fixed. $n = |X|$

```

1: procedure SPREADLB( $X, M, v, s$ )
2:    $L, R \leftarrow \underline{M}, \overline{M}$ 
3:   while  $L < R$  do
4:      $m_L \leftarrow L + \lfloor (R - L)/2 \rfloor$ 
5:      $m_R \leftarrow m_L + 1$ 
6:      $V_L, V_R \leftarrow LBV'(X, m_L/n), LBV'(X, m_R/n)$ 
7:     if  $\min(V_L, V_R) = 0$  then return true
8:     if  $V_L = V_R$  then
9:        $V, m \leftarrow V_L, m_L$ 
10:    break
11:    if  $V_L < V_R$  then
12:       $V, m, R \leftarrow V_L, m_L, m_L$ 
13:    else if  $V_L > V_R$  then
14:       $V, m, L \leftarrow V_R, m_R, m_R$ 
15:     $V_s \leftarrow \lfloor V \times s \rfloor$ 
16:    if  $V_s \leq \underline{v}$  then return true
17:     $EX \leftarrow \{ \llbracket x_i \leq \bar{x}_i \rrbracket \mid \bar{x}_i < m/n \} \cup \{ \llbracket x_i \geq \underline{x}_i \rrbracket \mid m/n < \underline{x}_i \}$ 
18:    if  $m = \overline{M}$  then  $EX \leftarrow EX \cup \{ \llbracket M \leq \overline{M} \rrbracket \}$ 
19:    if  $m = \underline{M}$  then  $EX \leftarrow EX \cup \{ \llbracket M \geq \underline{M} \rrbracket \}$ 
20:    if  $V_s > \overline{v}$  then
21:      return  $\llbracket v < V_s \rrbracket \wedge \bigwedge_{l \in EX} l \rightarrow false$ 
22:    else
23:      return  $\bigwedge_{l \in EX} l \rightarrow \llbracket v \geq V_s \rrbracket$ 

```

Hence, we can use binary search on the values m in M to find a minimum value for LBV' . Thus, only the mean values $\nu = \frac{m}{n}$ with $\underline{M} \leq m \leq \overline{M}$ have to be considered. This follows from the proof of Lemma 3 since the use of ν value is set to the mean of an arbitrary assignment.

3.2 Algorithm and Clause Learning Explanations

Algorithm 1 defines our approach to finding and asserting a lower bound on (scaled) variance. Note that when all $x \in X$ are fixed, we can simply calculate the actual variance in $\mathcal{O}(n)$ time.¹ It returns a clause which is a consequence of the constraint, and whose right-hand side gives the new propagation of the lower bound, otherwise it returns the vacuous *true* clause.

The algorithm searches through the integer range $L..R$ of values for the sum m that results in the minimum value for $LBV'(X, m/n)$. We compute the values V_L of LBV' for the (integer) midpoint m_L of $L..R$ and V_R for the position one to the right of m_L , namely m_R . If either of V_L or V_R give 0, then the overall variance bound is 0 and hence we return, since no propagation is possible. If V_L and V_R are equal, then we have found the lowest value and break from the loop, since two variance-equal and neighbouring points can only occur if they are both minima – this follows from convexity. Otherwise, we use the relative values of V_L and V_R to decide which half of the interval to keep. We store the best value found V and the sum value m where it occurs and update the appropriate bound L or R . If the interval ever shrinks to a singleton, then the loop exits. We compute the lower bound V_s on the scaled version of the variance v . If the lower bound is already subsumed, then we return a trivial clause *true*. Otherwise, we collect the literals that will appear in the explanation for the bound change or failure in EX . We collect all the lower and upper bounds that appear in

¹ In this case we still scale and round down, for consistency.

the ν -centred valuation, but not the bounds of the overlapping variables, in the explanation. If the optimal m value resides at one of the extremes of the sum M , then we collect the appropriate literal. Finally, if the new bound will cause unsatisfiability, then we add in the current upper bound for v and return an explanation for failure. Otherwise, we return an explanation for the new bound $v \geq Vs$.

This algorithm runs in $\mathcal{O}(n \log d)$ time, where $n = |X|$ and $d = \overline{M} - \underline{M}$. We can assume this algorithm is $\mathcal{O}(n \log n)$ when d is of size $\mathcal{O}(n^k)$ for any reasonably small constant k .

► **Lemma 5.** *Algorithm 1 returns a clause which is a consequence of $\text{spread}(X, M, v, s)$.*

Proof. The result clearly holds for the trivial clause *true*. The correctness of the value $V = \text{LBV}'(X, m/n)$ computed by Algorithm 1 follows from Lemma 3 and Lemma 4. We now show that the explanation collected in EX is correct, i.e., $EX \rightarrow s \times \sigma^2 \geq \lfloor s \times V \rfloor$ is universally true, which is the clausal explanation in both non-trivial cases. Assume to the contrary that there is a solution η of $\text{spread}(X, M, v, s)$ satisfying EX that has smaller variance. We can assume η is ν -centred for some ν and bounds on X (not necessarily the current bounds) [13]. But then η must set all x_i variables not appearing in EX to ν , otherwise there is an assignment allowed by EX which would be better. But η as a function of ν is identical to $\text{LBV}'(X, \nu)$ around the minima m/n . That is, all variables with upper bound below m/n are set to their upper bound, which is part of EX , and similarly for variables with lower bound above m/n , and the remaining variables take value ν . If m is not the upper or lower bound of the sum M , then m/n is a local (and global) minima of $\text{LBV}'(X, \nu)$, and hence also of η . If m is the upper or lower bound of M , then this bound is included in EX , and again m/n is the minima for $\text{LBV}'(X, \nu)$ and η . Contradiction. ◀

► **Example 6.** Consider an execution of the algorithm where $X = [x_1, x_2, x_3, x_4]$ with current domains $[0, 0.4, 2..3, 4..6]$, M has domain $6..10$, v has domain $0..400$ and $s = 100$. We start with $L = 6$, $R = 10$. Then $m_L = 8$ and $m_R = 9$. We compute $V_L = \text{LBV}'(X, 2) = 2$ and $V_R = \text{LBV}'(X, 2.25) = 2.015625$. We set $V = V_L$, $m = 8$ and $R = 8$. We compute $m_L = 7$ and $m_R = 8$, and compute $V_L = \text{LBV}'(X, 1.75) = 2.0123$ and $V_R = \text{LBV}'(X, 2) = 2$. We set $V = V_R$, $m = 8$ and $L = 8$. We exit the loop, computing $Vs = 200$. We collect the explanation $EX = \{\llbracket x_1 \leq 0 \rrbracket, \llbracket x_4 \geq 4 \rrbracket\}$ returning the explanation clause $\llbracket x_1 \leq 0 \rrbracket \wedge \llbracket x_4 \geq 4 \rrbracket \rightarrow \llbracket v \geq 200 \rrbracket$. Note that we do not collect $\llbracket x_3 \geq 2 \rrbracket$. This is because if the domain of x_3 is extended arbitrarily much in either or both directions, then it would still overlap with 2, and if it were reduced arbitrarily much from either or both directions, then the resulting minimal variance would increase. Thus, the lower bound holds regardless of the value of x_3 . ◀

4 The Gini Coefficient

Suppose we have an array X of n variables x_1, \dots, x_n . Define the Gini constraint $\text{gini}(X, M, g, s)$ to constrain M to be the sum of X (i.e., $\sum_{i=0}^n x_i$) and $g = \lfloor \text{Gini}(X) \times s \rfloor$ to be the Gini coefficient variable expressed as an integer scaled by s and rounded down. We can define this as a decomposition in MiniZinc as follows:

```

1 predicate gini(array[int] of var int:X, var int:M, var int:g, int:s) =
2   let { int: n=length(X);
3         int: range_size = ub_array(X) - lb_array(X);
4         array[int] of var 0..range_size: diffs
5           = [ abs(X[i]-X[j]) | i,j in index_set(X) where i<j];
6         var 0..s*(n div 2 + 1)*(n div 2)*(range_size): tot_diff
7           = sum(diffs) * s;
8         var 0..ub(tot_diff) div n: result_ = tot_diff div n;
9         var 0..ub(result_): result = result_ div M }

```

```
10  in v = result /\ M = sum(X);
```

Note that we try to bound the initial domains reasonably tightly, to reduce overhead in the solver. The function `lb_array` (and `ub_array`) returns the least (and greatest) possible value known *at compile time* of an array of decision variables. The upper bound on the sum of differences of numbers in the range $\min X.. \max X$ is to have half at each end, thus $\lceil \frac{n}{2} \rceil \times \lfloor \frac{n}{2} \rfloor \times (\text{range_size})$ where *range_size* is the difference between the minimum and maximum possible values of X .

Before going into the details of the propagator algorithm, we will prove a lemma that will help us. The formulation from Equation (2) will be used, since it is faster to calculate for ν -centred assignments and will simplify the proofs in this section. We show that the minimum Gini coefficient of the problem arises for a ν -centred assignment.

► **Lemma 7.** *For a given fixed mean value μ of variables X , a ν -centred assignment leads to the minimal value of the sum of absolute values of pairwise differences.*

Proof. We prove the lemma by contradiction. Assume w.l.o.g., that a non- ν -centred assignment ϕ for X is presented in non-decreasing order, i.e., $\phi(x_i) \leq \phi(x_{i+1})$ for $1 \leq i < n$, with the mean $\mu = \frac{1}{n} \sum_{i=1}^n \phi(x_i)$, and let $A = \sum_{i=1}^n (2i - n - 1)\phi(x_i)$ be the sum of the absolute values (by Equation (2)). Assume to the contrary that ϕ leads to a minimum sum A . Since the mean μ is fixed, we need not analyse the denominator of the Gini coefficient.

At least one variable must be unfixed, otherwise ϕ must be ν -centred. There must be at least one variable x_j with $\phi(x_j) < \bar{x}_j$, otherwise ϕ is ν -centred with $\nu = \max_{i=1}^n \phi(x_i)$. Furthermore, there must be at least one variable x_k with $\underline{x}_k < \phi(x_k)$ (analogously) and $1 \leq j < k \leq n$ (otherwise ϕ must be ν -centred). W.l.o.g., we can assume that $\phi(x_j) < \phi(x_{j+1})$ and $\phi(x_{k-1}) < \phi(x_k)$, because one can reorder the variables having the same value in ϕ , so that indices j and k are the greatest and least one with the values $\phi(x_j)$ and $\phi(x_k)$, respectively.

We construct a new assignment ϕ' for X , having the same mean as ϕ , as follows: $\phi'(x_i) = \phi(x_i)$ for $1 \leq i \leq n$ with $i \notin \{j, k\}$, $\phi'(x_j) = \phi(x_j) + \delta$, and $\phi'(x_k) = \phi(x_k) - \delta$ where $\delta \in \mathbb{R}$ and $\delta > 0$ is chosen, so that $\phi(x_j) + \delta \leq \min\{\phi(x_{j+1}), \bar{x}_j\}$ and $\max\{\phi(x_{k-1}), \underline{x}_k\} \leq \phi(x_k) - \delta$. Note that ϕ' is also in non-decreasing order, thus, its sum of absolute values is $A' = \sum_{i=1}^n (2i - n - 1)\phi'(x_i)$ by Equation (2). If A is the minimal sum of absolute values then $0 \geq A - A'$. Since ϕ and ϕ' only differ in the variable values for x_j and x_k , it holds $0 \geq (2j - n - 1)(\phi(x_j) - \phi'(x_j)) + (2k - n - 1)(\phi(x_k) - \phi'(x_k))$, which is $0 \geq (2j - n - 1)(-\delta) + (2k - n - 1)\delta = 2(k - j)\delta$. Due to $j < k$, we have $2(k - j)\delta > 0$, which contradicts the assumption that A is the minimal sum of absolute values. ◀

Since fixing the mean fixes the divisor of the Gini coefficient, we have the following obvious consequence. Given a fixed mean μ , the ν -centred assignment θ_ν where $\mu = \frac{1}{n} \sum_{i=1}^n \theta_\nu(x_i)$ leads to minimal Gini coefficient. And this must hold for any mean.

► **Corollary 8.** *A Gini-minimal assignment of X must be ν -centred.*

4.1 Lower Bound on Gini Coefficient

In this section we present a log-linear time propagator for filtering the lower bound of the Gini coefficient using binary chop.

We will now, similarly to the approach used for variance, show how to calculate a lower bound on the Gini Coefficient. The key to this is, again, finding the best ν -centred assignment, which we know must be a lower bound.

Let $L = \min_{i=1}^n \underline{x}_i$ be the least lower bound and $U = \max_{i=1}^n \bar{x}_i$ be the greatest upper bound, then we can define a real-value relaxed lower bound on the Gini coefficient of X as

$$LBG(X) = \min_{\beta=L}^U Gini(X, \beta), \quad \text{where} \quad Gini(X, \nu) = \frac{\sum_{i=1}^n \sum_{j=i+1}^n |\theta_\nu(x_i) - \theta_\nu(x_j)|}{n \sum_{i=1}^n \theta_\nu(x_i)}.$$

By Equation (2) and given a non-decreasingly sorted list of the bounds of X , we can calculate $Gini(X, \nu)$ in linear time over $|X|$. Next, we show that we do not have to consider all values between L and U .

► **Lemma 9.** *The minimum Gini coefficient occurs at a ν -centred assignment where $\nu \in \cup_{i=1}^n \{\bar{x}_i, \underline{x}_i\}$.*

Proof. By Corollary 8, we know that the minimum Gini coefficient occurs with a ν -centred assignment. We need to show that there are no local minima in any segment between two consecutive bounds, which, if true, will allow us to disregard any point that is not a bound of one of the variables when searching for the lower bound on Gini. Consider a segment between two consecutive bounds l and u . Since no other bound occurs in between l and u , the three sets of variables in X that will be set to their upper bounds, their lower bounds, and to ν , respectively, will remain constant across all ν -centred assignments when $l \leq \nu \leq u$. Assume these variables in X , and what they are set to, occur sorted such that $B = 1..j-1$ are the indices of the variables below the segment (i.e., where $\bar{x}_i \leq l$), the indices $C = j..j+k-1$ are of the variables that cover the segment (i.e., where $\underline{x}_i \leq l \leq u \leq \bar{x}_i$), and $A = j+k..n$ are the indices of the variables above the segment (i.e., where $x_i \geq u$). Using Equation (2), the formula for the Gini coefficient across this segment $G(\nu)$ is hence

$$G(\nu) \equiv \frac{\sum_{i \in B} (2i - n - 1) \bar{x}_i + \sum_{i \in C} (2i - n - 1) \nu + \sum_{i \in A} (2i - n - 1) \underline{x}_i}{n (\sum_{i \in B} \bar{x}_i + \sum_{i \in A} \underline{x}_i + k\nu)}$$

Let $D = \sum_{i \in B} (2i - n - 1) \bar{x}_i + \sum_{i \in A} (2i - n - 1) \underline{x}_i$, $N = \sum_{i \in B} \bar{x}_i + \sum_{i \in A} \underline{x}_i$ and $c = \sum_{i \in C} (2i - n - 1)$. Differentiating $G(\nu)$ w.r.t. to ν , we get the following by the quotient rule

$$\frac{\partial}{\partial \nu} \frac{D + c\nu}{n(N + k\nu)} = \frac{cn(N + k\nu) - (D + c\nu)nk}{n^2(N + k\nu)^2} = \frac{cN - kD}{n(N + k\nu)^2}$$

Note that the sign of the slope is determined by the numerator, and does not depend on ν . Hence there can be no local minima in the segment. ◀

From the above result, it follows that we need not even consider the real-relaxed case for Gini, as all values ν of importance are integers. Next, we prove that we can use binary chop to find the ν -centred assignment that minimises the Gini coefficient.

► **Lemma 10.** *Going through the segments of LBG in increasing order of the sorted end-points, a positive slope is not (directly or indirectly) followed by a negative slope.*

Proof. Let us revisit the derivative of each segment, i.e., $\frac{cN - kD}{n(N + k\nu)^2}$. The sign of its slope is determined by the numerator $cN - kD$. We can rewrite c to a closed form formula because it is an arithmetic sum. We get $k \frac{((2j-n-1) + (2(j+k-1)-n-1))}{2}$ which simplifies to $k(k+2j-n-2)$. Rewriting the numerator we get $k(k+2j-n-2)N - kD$, which we can simplify to $k((k+2j-n-2)N - D)$. Because k is always non-negative, the outer k cannot determine the sign of the slope, only its magnitude, thus we can safely ignore it for our purposes. Ignoring the outer k and expanding N and D we get $(\sum_{i \in B} (k+2j-n-2)\bar{x}_i - (2i-n-1)\bar{x}_i) + (\sum_{i \in A} (k+2j-n-2)\underline{x}_i - (2i-n-1)\underline{x}_i)$. This simplifies to $\sum_{i \in B} (k+2j-2i-1)\bar{x}_i +$

■ **Algorithm 2** Only run if not all fixed. $n = |X|$

Require: $B = [\beta_1, \dots, \beta_{2n}]$ is a sorted array of bounds of X

```

1: procedure GINILB( $X, M, g, s$ )
2:    $L, R \leftarrow 1, 2n$ 
3:   while  $L < R$  do
4:      $l \leftarrow L + \lfloor (R - L)/2 \rfloor$ 
5:      $r \leftarrow l + 1$ 
6:      $G_l \leftarrow \text{Gini}(X, \beta_l)$ 
7:      $G_r \leftarrow \text{Gini}(X, \beta_r)$ 
8:     while  $G_l = G_r$  do
9:       if  $l > L$  then
10:         $l, G_l \leftarrow l - 1, \text{Gini}(X, \beta_{l-1})$ 
11:       else
12:         $L \leftarrow r$ 
13:       break
14:     if  $G_l < G_r$  then
15:        $R \leftarrow l$ 
16:     else
17:        $L \leftarrow r$ 
18:    $m, G \leftarrow R, \text{Gini}(X, \beta_R)$ 
19:    $Gs \leftarrow \lfloor G \times s \rfloor$ 
20:   if  $Gs \leq g$  then return true
21:    $EX \leftarrow \{ \lfloor x_i \leq \bar{x}_i \rfloor \mid \bar{x}_i \leq \beta_m \} \cup \{ \lfloor x_i \geq \underline{x}_i \rfloor \mid \beta_m \leq \underline{x}_i \}$ 
22:   if  $Gs > \bar{g}$  then
23:     return  $\llbracket g < Gs \rrbracket \wedge \bigwedge_{l \in EX} l \rightarrow \text{false}$ 
24:   else
25:     return  $\bigwedge_{l \in EX} l \rightarrow \llbracket g \geq Gs \rrbracket$ 

```

$\sum_{i \in A} (k + 2j - 2i - 1)x_i$. Let us denote the first term by T_B and the second term by T_A . The maximum value of $i \in B$ is $j - 1$; thus, all terms of T_B are non-negative; and the minimum value of $i \in A$ is $j + k$; thus, all terms of T_A are non-positive. At any segment where the slope is positive, we must have that $T_B > |T_A|$. Moving to the next segment variables could move from A to the overlap part, or from the overlap part to B , or both. Let us consider two cases: (1) at least one variable moves from A to the overlap and (2) at least one variable moves from the overlap to B .

Case (1): In this case, $|T_A|$ will decrease because at least one negative term will be removed, and k will increase by at least one. And T_B will increase because k will increase by at least one.

Case (2): In this case, $|T_A|$ will decrease because $k + 2j$ will increase by at least one (while k decreases, j must increase the same amount) and T_B will increase because at least one more positive term will be introduced and $k + 2j$ will increase by at least one.

As a result, moving from one segment where $T_B > |T_A|$ to the next, we must maintain that $T_B > |T_A|$. Hence, once the slope is positive it cannot become negative. ◀

► **Corollary 11.** *LBG has no local minima that is not a global minima.*

Given this result we can again use binary chop to find the global minimum of $\text{Gini}(X, \nu)$.

4.2 Algorithm and Clause Learning Explanations

The algorithm for computing a lower bound on the (scaled) Gini coefficient is given in Algorithm 2. It does a binary chop across the sorted list of bounds of the x variables: $\beta_1, \dots, \beta_{2n}$. L and R hold the left and right bound of indices into this array, for where the *rightmost* minimum lies. We compute the midpoint l of L and R and its neighbour r and

compare the Gini values. If they are identical in value, we extend the interval $l..r$ to the left until we find a difference, or fill the entire region from L to r , in which case we chop and set $L = r$. We can incrementally compute $Gini(X, \beta_{i-1})$ from $Gini(X, \beta_i)$ in constant time with a little care. Otherwise we update the L or R to keep the rightmost minimum between them. The loop terminates when $L = R$. We calculate the Gini value G , and its scaled version Gs . Note that we use the *rightmost* ν -value when we find a range of bounds with the same Gini value. If the new lower bound is not stronger than the current bound we return a *true* clause. Otherwise we collect as explanation: all lower bounds where x is less than *or equal to* the ν value β_m , and all upper bounds where x is greater than *or equal to* the ν -value β_m . If the new bound causes a violation we return an explanation for the violation, otherwise we return (an equivalent clause) explanation for the new bound. Overall the algorithm is $O(n \log n)$ since the initial sorting is of this complexity, the while loop can only execute $O(\log n)$ times, and the *Gini* calculations are linear (using Equation (2)).

Note also that the scaling and rounding to integer takes place after the binary chop has terminated. This is important, since not having the best accuracy when calculating the slopes can lead to the algorithm terminating with a solution that is only optimal given the rounding to integer. Using this can then lead to incorrect explanations.

► **Lemma 12.** *Algorithm 2 returns a clause which is a consequence of $gini(X, M, g, s)$.*

Proof. The result clearly holds for the trivial clause *true*. The correctness of the value G computed by Algorithm 1 follows from Lemma 7, Lemma 9 and Lemma 10. We now show that the explanation collected in EX is correct. Assume to the contrary that there is a solution η of $gini(X, M, g, s)$ satisfying EX that has Gini coefficient $G' < G$. We can assume η is ν -centred for some ν and some bounds on X (not necessarily the current bounds) by Lemma 7. But then η must set all x_i variables not appearing in EX to ν , otherwise there is an assignment allowed by EX which would be no worse. So we can consider η as defined by ν and the bounds appearing in EX . Hence, it has the same properties as the $Gini(X, \nu)$ but with fewer bounds: importantly it has negatively sloped and flat segments, followed by positively sloped and flat segments. If we move ν smoothly up from β_m then because the slope of the curve $G(\nu)$ defined in Lemma 9 is positive by construction (this is why it is important to take the rightmost point with the least Gini value), and this slope only depends on the bounds in EX , the Gini value for η must increase. If we move ν smoothly from β_m downwards then the slope on $G(\nu)$ defined in Lemma 9 is either negative or zero by construction. Since the slope only depends on the bounds in EX , if it is negative the Gini value of η must increase. If it is zero then the Gini value must stay the same. This holds until we cross a lower bound in x_i not in EX , until then the Gini values for η are the same as $Gini(X, \nu)$. But once we cross this bound the sum defined by solution η is smaller than that for $Gini(X, \nu)$ and overall the Gini value must be larger. Contradiction. ◀

5 Experimental Evaluation

For both dispersion constraints we use four configurations: The decomposition (Dcmp); the simple minimal propagator (Simp), which only propagates on the dispersion once all variables X are fixed; the binary chop lower-bound propagator with the proposed (LB) and trivial (LB-TL) nogood learning. The trivial learning simply uses all bounds of all the variables as explanation. We implemented the above propagators in Chuffed and ran the below experiments using MiniZinc [8] on an Intel Xeon 8260 CPU (24 cores) with 268.55 GB of RAM. A single core, 8 GB of RAM, and a 20 minute timeout was allocated for the solving of each instance.

■ **Table 1** Summary of the results of the propagators for the 195 dispersion-only instances

prob	prop	proved			best			no solution	
		total	first	sole	total	first	sole	total	sole
Spread	Dcmp	38	17	2	43	22	7	127	127
Spread	Simp	25	2	0	85	60	58	0	0
Spread	LB-TL	38	12	0	59	30	14	0	0
Spread	LB	40	25	3	115	96	71	0	0
Gini	Dcmp	33	11	0	37	17	6	97	97
Gini	Simp	25	10	0	171	154	144	0	0
Gini	LB-TL	40	15	0	42	32	11	0	0
Gini	LB	47	40	6	26	20	2	0	0

5.1 Dispersion Only

Let's first start off with a simple problem to get a good understanding of the general performance. We have n variables $x_1, \dots, x_n \in X$, each with domain $l_i..u_i$. For each i , two integers are uniformly randomly drawn from the range $-n^2..n^2$ (if equal, the second gets redrawn until not equal). The lower number becomes l_i and the higher becomes u_i . For each $n \in \{2, 3, \dots, 40\}$, we create five instances, resulting in $5 \times 39 = 195$ instances. The goal is to minimise dispersion. Since Gini is ill-defined for negative values, all numbers of all Gini instances are uniformly incremented (if needed) until the least one reaches 1.

For these simple benchmarks using learning with the global propagators simply adds overhead, since the search is so simple that little is repeated. Learning does improve the decomposition though, since it can learn on intermediate variables. We show the results for the learning versions since we expect those to be used in real benchmarks.

The results are shown in Table 1. The table shows, for each method, the *total* number of instances where it *proved* optimality, the number of times the algorithm proved optimality *first* on an instance, and the number of instances where it was the *sole* algorithm to prove optimality. It then shows for all instances the *total* number of instances where the algorithm found the *best* solution of any found, the number of instances where it did this *first* of all algorithms, and the number of times it was the *sole* method to find this best solution. Finally we show the *total* number of instances where the method found *no solution*, and for how many it was the *sole* method to do so. Clearly for **spread**, LB is the best method in all categories. It finds the best solution for all instances, and is fastest on almost all. The decomposition fails on most instances because the sizes of the intermediate values it computes with get quickly too large for the solver to deal with. For **gini**, perhaps somewhat surprisingly, the simple propagator is best at finding solutions. For Gini, the best solutions are usually found early in the search, so the fast simple propagator will find good solutions very quickly compared to more expensive propagators LB and LB-TL. For **gini** the size of intermediate values is smaller, since numbers are not squared. The decomposition is competitive in terms of optimality, but still fails on many instances.

5.2 Job-Shop Scheduling

The dispersion only problems are not complex enough to illustrate why we need learning for dispersion propagators. Let us, instead, run experiments on a type of problem where using a learning solver is often desirable, namely, a scheduling problem.

Consider a job-shop scheduling problem but with multiple agents. The goal is to schedule a set of jobs J on a set of machines M . Each job $j \in J$ consists of one task per machine,

where each task can only be processed by a given machine and takes a given amount of time to be processed. A machine can only process one task at a time, and the tasks within each job have to be processed in order (task t must finish before task $t + 1$ starts). In addition, we have a set of agents A that submit these jobs. More formally, each job $j \in J$ belongs to exactly one agent $a \in A$.

We consider a limited horizon (after which scheduling jobs is pointless due to zero utility) with optional jobs (because not all jobs will fit within the limited horizon). The utility of a given agent is the machine-hours used by their jobs. We further assume that each agent submits more jobs than can be accommodated. This results in comparable utility functions.

The *objective* of the overall problem is either to maximise efficiency (use as many machine-hours as possible) or minimise dispersion (give similar amounts of machine-hours to each agent), respectively denoted **MaxEff** and **MinDis**. In either case, a bound on the other metric is used to ensure some solution quality in the other aspect. For example, giving no one anything is a minimal dispersion schedule, but not very efficient.

We use the large instances (30 or more jobs) of [18] (resulting in 50 instances),² and for each instance split the jobs uniformly across n agents, for every n such that each agent has between 10 and 20 jobs (resulting in $20 \times 2 + 20 \times 3 + 10 \times 6 = 160$ instances). We perform this uniform split three times to generate a more diverse instance space (resulting in $160 \times 3 = 480$ instances). We run these on the two configurations (**MaxEff** and **MinDis**); the resulting number of instances to test each propagator with is 960.

We first run, with a 20 minute time-out for each instance, a variation of the problem where the objective is to minimise the makespan when all jobs are scheduled. The result of this will be a good indication of how long to set our limited horizon for each instance such that no agent can fit all their jobs. We set the limited horizon for the instance in question to be half of this minimum makespan.

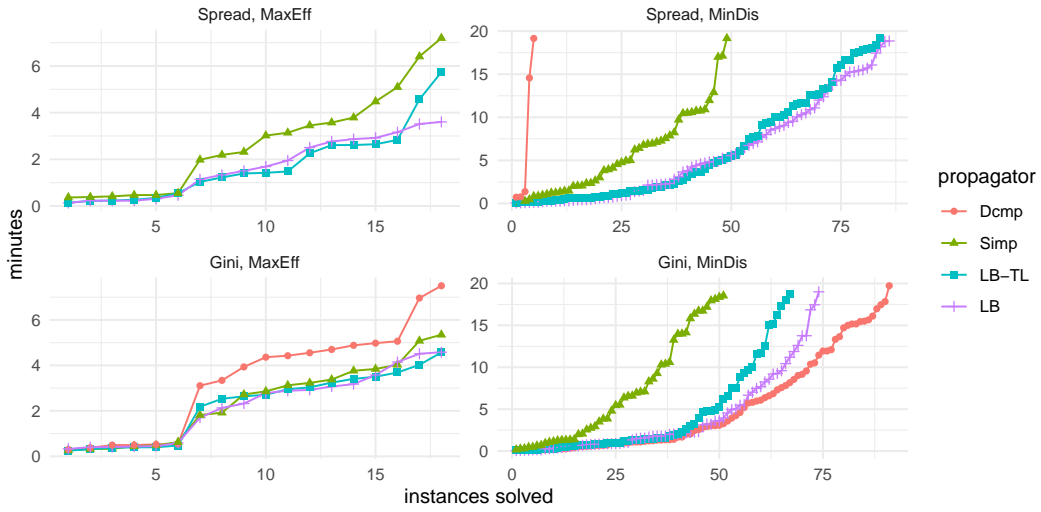
Next, we run, again with a 20 minute time-out for each instance, a variation of the problem where the objective is to maximise machine-hour utilisation given the limited horizon. This will give a good indication of the optimal utilisation possible for each instance. We will then use this number H to calculate our machine-hour utilisation bounds and dispersion bounds. For spread we use scale 1, because the numbers are already large, and for Gini we use scale 10000. For both spread and Gini with **MinDis** we use $\lceil H \times 0.8 \rceil$ as a lower bound on efficiency, allowing 20% slack on efficiency; for spread with **MaxEff** we use $\lfloor (0.2 \times H/|A|)^2 \rfloor$ as an upper bound on spread, allowing a 20% standard deviation from the per-agent average machine-hour utilisation. For Gini and **MaxEff** we use 2000 as an upper bound on Gini, allowing a 20% unequal Gini distribution.

The cactus plots in Figure 1 show the number of problems solved to optimality at different times for the different variations: using the spread problems on top, the Gini problems on bottom, the **MaxEff** problems on the left, and the **MinDis** problems on the right. In general, the **MaxEff** problems are much harder than the minimisation problems.

Clearly for **spread** the decomposition is very weak, this is because the size of intermediates involved is very large, and often goes outside the range that solvers can handle. The simple propagator is much better, and the full spread propagators much better again. Interestingly our small explanations are only slightly more effective than the trivial explanations.

The results for **gini** are quite different. The decomposition is actually capable of proving

² There are 20 instances with 30 jobs (the first half of which have 15 machines and the rest have 20 machines), 20 instances with 50 jobs (again, the first half of which have 15 machines and the rest have 20 machines), and 10 instances with 100 jobs, all with 20 machines.



■ **Figure 1** Cactus plot of instances solved to proof of optimality of the propagators for jobshop

■ **Table 2** Summary of the results of the propagators for the 960 jobshop instances

prob	prop	proved			best			no solution	
		total	first	sole	total	first	sole	total	sole
Spread	Dcmp	5	3	0	10	8	5	927	904
Spread	Simp	67	14	0	270	137	100	15	0
Spread	LB-TL	102	48	12	433	250	148	18	0
Spread	LB	104	53	16	695	558	412	18	0
Gini	Dcmp	109	46	15	264	123	82	104	86
Gini	Simp	69	11	0	249	130	108	16	1
Gini	LB-TL	85	25	1	345	197	131	19	0
Gini	LB	92	30	2	620	500	402	19	0

optimality more than other methods on the minimisation problems. This is because it can use learning on the intermediate variables in the decomposition. Having a richer language of learning can often improve proofs of optimality/unsatisfiability. The propagator with our small explanations is the second best. For the **MaxEff** problems, the decomposition is performing the worst and the other methods are roughly equivalent.

For a more detailed comparison we examine Table 2 which is organised like Table 1. Table 2 clearly illustrates the power of the global propagators. For **spread**, the propagator with our small explanations dominates all methods. For **gini**, while the decomposition is best for proving optimality, overall, it is much weaker than the global propagator, which finds the best solution in over twice as many instances, and almost always finds a solution. It also more clearly illustrates the importance of small explanations, LB is far better than LB-TL in terms of best solutions. We also ran with no learning on all four methods, but it simply timed out on all instances.

6 Related Work

The spread constraint was introduced by [13]. Many previous methods assume that the mean is fixed, e.g. [17]. This is no impediment when the problem is about allocating a given set

of resources without mutual exclusions and without uncertainties, but not for the general case where the amount of resource available, or the amount that can be used is not fixed. [12] introduce a domain consistent propagator for dispersion constraints (including spread) but again assume a fixed/given mean. While an extension for a bounded (variable) mean is proposed, the main criticism is that that algorithm basically runs the domain propagator for each value in the mean, essentially adding another factor to the time complexity. The resulting time complexity is essentially $\mathcal{O}(|X|^3 \cdot |D(X)| \cdot |D(\mu)|)$, which is impractical in many cases. An MDD-formulation of the spread constraint is presented in [11]. They, again, assume a fixed mean. They extend their algorithm to work on variable means, but do not fully answer the question for unknown means. Their use of a probability density function cannot guarantee correct propagation when the mean is fully unknown.

An earlier paper by [13] introduces a bounds consistent propagator for the spread constraint, which covers mean, median, and standard deviation (and thus indirectly variance). They assume finite-domain variables X and bounded continuous domains for mean and standard deviation. They introduce two propagators, one which runs in $\mathcal{O}(|X| + |D(X)|^2)$ time and one (bounds consistent) that runs in $\mathcal{O}(|X|^2)$. One criticism (posed in [17] which have one author in common) is that the latter has never been implemented, is hard to understand, and contains mathematical errors. Thus, we do not compare against it. The former assumes that $|X|$ dominates $|D|^2$, but it does not hold in our use cases. In our jobshop instances, $|X|$ (between 2 and 10) is much smaller than $|D(X)|$ (between 0 and 47186).

Propagators for several other statistical constraints have been introduced in the literature. These include, the deviation constraint [16], mean, median, and weighted mean constraints [2], an occurrence balancing constraint [1], the two-sum constraint [7] (which subsumes spread), and many other constraints [15]. [10] provide a review of when to use what balancing/dispersion constraint, depending on the nature of the situation.

Note that in all of the spread propagators above, none generate explanations of their propagation, as required if we want to use them in a learning CP solver [9].

To the best of our knowledge, we are not aware of any propagators for the Gini coefficient. Bounds on the Gini coefficient have been established in the statistics literature [5, 6]; however, these approaches concern obtaining bounds on the actual Gini coefficient of an existing, large population from relatively few samples, and not about filtering possibilities of an unknown population.

7 Conclusion

In this paper we define efficient propagators for propagating the lower bound on variance and the Gini coefficient that also generate explanations of their propagations, allowing them to be used in learning CP solvers. Propagating the lower bound is the critical case to consider for measures of dispersion, since the typical requirement is to bound or minimize dispersion to generate fair solutions. We do not consider propagating the bounds of the X variables being measured, since preliminary experimentation indicated this happens very late in the search tree, and hence cannot lead to significant speedups. Note however that if we stored the explanations generated by the propagator as clauses, some propagation on the X variables could occur.

References

- 1 Christian Bessiere, Emmanuel Hebrard, George Katsirelos, Zeynep Kiziltan, Émilie Picard-Cantin, Claude-Guy Quimper, and Toby Walsh. The balance constraint family. In Barry

- O’Sullivan, editor, *CP’14*, pages 174–189. Springer, 2014.
- 2 Alessio Bonfietti and Michele Lombardi. The weighted average constraint. In Michela Milano, editor, *CP’12*, pages 191–206. Springer, 2012.
 - 3 Geoffrey Chu. *Improving Combinatorial Optimization*. PhD thesis, Department of Computing and Information Systems, University of Melbourne, Australia, 2011.
 - 4 Philip M. Dixon, Jacob Weiner, Thomas Mitchell-Olds, and Robert A. Woodley. Bootstrapping the Gini coefficient of inequality. *Ecology*, 68:1548–1551, 1987. (Erratum in *Ecology*, 69:1307, 1987).
 - 5 Joseph L Gastwirth. The estimation of the Lorenz curve and Gini index. *The review of economics and statistics*, pages 306–316, 1972.
 - 6 Farhad Mehran. Bounds on the Gini index based on observed points of the Lorenz curve. *Journal of the American Statistical Association*, 70(349):64–66, 1975.
 - 7 Jean-Noël Monette, Nicolas Beldiceanu, Pierre Flener, and Justin Pearson. A parametric propagator for pairs of sum constraints with a discrete convexity property. *AIJ*, 241:170–190, December 2016. doi:10.1016/j.artint.2016.08.006.
 - 8 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *CP’07*, LNCS 4741, pages 529–543. Springer, 2007.
 - 9 Olga Ohrimenko, Peter Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14:357–391, 2009.
 - 10 Philippe Olivier, Andrea Lodi, and Gilles Pesant. Measures of balance in combinatorial optimization. *4OR*, June 2021. doi:10.1007/s10288-021-00486-x.
 - 11 Guillaume Perez and Jean-Charles Régin. MDDs are efficient modeling tools: An application to some statistical constraints. In Domenico Salvagnin and Michele Lombardi, editors, *CPAIOR’17*, pages 30–40. Springer, 2017.
 - 12 Gilles Pesant. Achieving domain consistency and counting solutions for dispersion constraints. *INFORMS J. Comput.*, 27(4):690–703, 2015. doi:10.1287/ijoc.2015.0654.
 - 13 Gilles Pesant and Jean-Charles Régin. SPREAD: A balancing constraint based on statistics. In Peter van Beek, editor, *CP’05*, LNCS 3709, pages 460–474. Springer, 2005. doi:10.1007/11564751_35.
 - 14 Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.
 - 15 Roberto Rossi, Özgür Akgün, Steven Prestwich, and S. Armagan Tarim. Declarative statistics, 2017. arXiv:1708.01829.
 - 16 Pierre Schaus, Yves Deville, and Pierre Dupont. Bound-consistent deviation constraint. In Christian Bessière, editor, *CP’07*, pages 620–634. Springer, 2007.
 - 17 Pierre Schaus and Jean-Charles Régin. Bound-consistent spread constraint. *EURO J. Comput. Optim.*, 2(3):123–146, 2014. doi:10.1007/s13675-013-0018-8.
 - 18 Eric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.