Acquiring and Selecting Implied Constraints with an Application to the BinSeq and Partition Global Constraints

J. Cheukam Ngouonou[1,2,4], R. Gindullin[1,2], C.-G. Quimper[4], N. Beldiceanu[1,2] and R. Douence[1,2,3]

[1]IMT Atlantique, Nantes, France [2]LS2N, Nantes, France
[3]INRIA, Nantes, France [4]Université Laval, Québec City, Canada
ramiz.gindullin@it.uu.se, nicolas.beldiceanu@imt-atlantique.fr,
jovial.cheukam-ngouonou.1@ulaval.ca, Remi.Douence@imt-atlantique.fr,
Claude-Guy.Quimper@ift.ulaval.ca

Abstract. We propose a machine-assisted approach to synthesise implied constraints for global constraints based on combinatorial objects. By reusing the Bound Seeker [8], we generate thousands of relationships between features. We present a scalable algorithm that automatically selects the relationships that filter the most, which we manually prove. We consider the Partition and the BinSeq constraints, which model the different ways of dividing a collection of objects into clusters, or the repartition of shifts in a 0–1 sequence. We use Partition and BinSeq in the Balanced Academic Curriculum Problem (BACP), and the Balanced Shift-Scheduling Problem (BSSP), where we optimise the distribution of the work to balance the workload. For 2 models of the BACP and 2 models of the BSSP, we show how the filtering inferred by the Bound Seeker improves the cost of the solution found on different solvers. This filtering proved optimality for all CSPLib instances of the BACP.

1 Introduction

Constraint solvers rely on Filtering Algorithms (FAs) that are designed for each constraint. In recent decades, an effort has led to the custom development of FAs [29,7,5] to reduce the search space. Synthesising FAs can be traced back to research on the Rabbit solver [23], and studies on synthesising propagators for low-arity constraints [17]. However, the need to consider the interaction of an increasing number of constraint parameters raises the question of how to synthesise FAs that can be used in different solvers. We propose a novel machine-assisted approach to generate a FA that consists of a set of bounds used for filtering a constraint for which a reformulation already exists. The Bound Seeker [8] discovers each bound, and our method chooses the most important ones that we manually prove after the selection process. We consider satisfaction or optimisation problems whose solution is given by a combinatorial object, e.g. a graph, a 0–1 sequence, and a partition. The constraints are imposed on the features of this object. e.g. in the balanced academic curriculum problem, one wants to

partition courses into terms. The solution is a partition that is constrained to spread the workload between terms. One can define a constraint optimisation problem whose objective is to minimise the sum of the squares of the partition sizes. Modelling this problem to obtain a strong filtering is not trivial.

The Bound Seeker [8,20] identifies conjectures of sharp bounds between the features of a combinatorial object. These conjectures are inequalities between a single feature and a function over other features, e.g. consider a partition of n elements where $\overline{M}$ is the size of the largest partition and S is the sum of the squares of the partition sizes. The Bound Seeker discovers the sharp bound $S \leq (n \bmod \overline{M})^2 - \overline{M}(n \bmod \overline{M}) + n\overline{M}$. This bound can be added, as a redundant constraint, to a model to enhance filtering. As our machine-assisted synthesis method is independent of bound generation, we treat the Bound Seeker [8] as a black box that takes combinatorial object instances as input and outputs valid inequalities, i.e. arithmetic constraints, for all input data.

The Bound Seeker can find thousands of relations between features, but not all of these bounds are equally useful for filtering. We designed an algorithm that selects a subset of these bounds that are relevant for filtering the variable domains of a constraint satisfaction problem whose solution is an instance of the combinatorial object. Finally, we add the inequalities associated with the bounds to the constraint model. The filtering performed by these inequalities acts as a filtering algorithm for a global constraint encoding the combinatorial object. To illustrate the efficiency of our method, we choose the partition and 0–1 sequence combinatorial objects introduced in [20]. Partitions are used in rostering problems to distribute workload between employees or in the balanced academic curriculum problem [22] to group courses of a same term. 0–1 sequences are used in nurse scheduling problems to assign shifts to nurses wrt to a set of rules. An objective is to obtain partitions or shifts of similar size. It is usual to approximate this objective by minimising the size of the largest partition or the longest shift, as filtering algorithms for this objective are efficient. More balanced solutions can be obtained by minimising the squared size of partitions or shift sizes, but this is more difficult, and few constraints exist. Spread [26,31] minimises the variance of a vector, equivalent to minimising the sum of squares when the mean is fixed. Deviation [27,30] computes the sum of absolute differences between a vector's values and their mean. In contrast, Partition aims to spread the number of times these values occur. Balance [11] minimises the difference between the most and least frequent values, as does Partition, while also minimising the squared number of occurrences. This is where exploiting the inequalities found by the Bound Seeker becomes useful. Our contributions include: (1) *Machine Assisted Implied Constraints Synthesis.* Based on a set of discovered bounds, we present a framework for generating FAs that can be easily integrated into multiple constraint solvers. (2) *Bound Selection.* We propose a method for selecting the most significant bounds automatically from a large set of candidates. (3) *Global Constraints for Balanced Solutions.* We improve the search for better balanced solutions to partitioning and shift scheduling problems by introducing the Partition and BinSeq constraints. (4) *Theoretical Insights.* We prove a

sharp lower bound on the sum of the squared partition sizes that penalises both results above and below the load average. The penalty increases quadratically, which helps to prevent large deviations.

Sect. 2 gives the background. Sect. 3 presents the framework for selecting redundant constraints. Sect. 4 defines the PARTITION and BINSEQ constraints. Sect. 5 uses the framework of Sect. 3 to generate redundant constraints for PARTITION and BINSEQ. Sect. 6 evaluates their impact on the BACP and BSSP.

2 Background

Presolvers Presolvers simplify models by fixing variables, reducing constraints, and strengthening bounds in mixed integer programs [2]. SAT solvers use techniques like unit propagation and subsumption [12]. Constraint satisfaction problem preprocessing is complex due to constraint diversity, but subexpression elimination enhances filtering [28]. Presolving can globally improve models, as seen in time series pattern analysis for bound inference [4]. Redundant constraints can strengthen filtering and reduce search space. Gent *et al.* [18] synthesise filtering algorithms for small constraints via exponential offline preprocessing. Charnley *et al.* [14] generate implied constraints for finite algebra. Our work differs in that their implied constraints have a simpler form, and their constraint selection process is costly, analysing all possible subsets of constraints up to a limit.

Conjecture Discovery Few systems search for bounds on the features of a combinatorial object. Among the best known are the S. Fajtlowicz's Graffiti program [16] and P. Hansen's AutoGraphiX system [3,21]. We describe a third, more recent system called the Bound Seeker [8] to exploit the sharp bounds it discovers, to synthesise implied constraints. Let $\mathcal{O}$ be a combinatorial object identified by m features $x_1, x_2, \ldots, x_m$ taking integer values, where the maximum value of the first feature x_1 restricts the range of $x_2, \ldots, x_m$. The Bound Seeker takes as input a set of extreme instances for $\mathcal{O}$ and learns inequalities of the forms $x_i \leq u_i(P)$ and $x_i \geq l_i(P)$, where l_i and u_i are symbolic functions and $P \subseteq \{x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_m\}$. Thus, there is a possibility of $m2^m$ unique bounds. These inequalities bound the feature x_i, and the Bound Seeker guarantees that these bounds are tight, i.e. there exists an instance of $\mathcal{O}$ for which the equality holds. These bounds, called *conjectures*, were derived from some instances of $\mathcal{O}$ and may not apply to all possible instances. Turning these conjectures into theorems requires a formal proof.

3 The Framework

The framework takes a combinatorial object and returns a set of constraints. They are computed only once, and later used to strengthen filtering. The Bounds Seeker breaks down the process into three steps. First, it starts with input instances of a combinatorial object and generates conjectures in the form of redundant constraints that enhance filtering. However, not all constraints result

in additional filtering. In Step 2, we remove some constraints to create a subset that filters as effectively as if all constraints were included. Two constraint sets are equivalent if replacing one set with another in a constraint model solves a problem with the same number of backtracks. In Step 3, we manually prove the conjectures underlying the selected constraints.

3.1 Generating Candidate Constraints

We start with a constraint satisfaction problem whose solution is expressed in terms of a combinatorial object, e.g. a graph, a 0–1 sequence, or a partition. We identify the features $x_1, x_2, \ldots, x_m$ of this combinatorial object that are important for the model. These features are numerical values that characterise the solution. For a graph, this could be the number of nodes, edges, components, etc. The problem's definition usually constrains these features.

The Bound Seeker [20] takes as input instances of a combinatorial object and a list of features that can be computed on these objects. It computes a set of conjectures $\mathcal{C} = \{C_1, \ldots, C_{|\mathcal{C}|}\}$. Each conjecture C_i is of the form $x_i \leq u_i(x_{j_1}, x_{j_2}, \ldots)$ or $x_i \geq l_i(x_{j_1}, x_{j_2}, \ldots)$. These inequalities could be added directly to the model of the constraint satisfaction problem to enhance filtering. Even if, in practice, the Bound Seeker is unable to find all $m2^m$ possible distinct conjectures in a reasonable time, it is nevertheless able to compute thousands of them. The time spent on filtering these constraints could outweigh the time saved by pruning the search tree. Many conjectures may filter the same values, and some conjectures might have their filtering dominated by others.

Alg. 2 is an algorithm that removes some conjectures on bounds from the set $\mathcal{C}$ without reducing the filtering. It returns a subset of conjectures that is minimal in the sense that removing any conjecture from this subset reduces the amount of filtering. Alg. 2 solves a Constraint Satisfaction Problem (CSP). Given the input $\mathcal{C}$ of bound conjectures, let $M(\mathcal{C})$ be a CSP outputting all possible combinations of values for the features $x_1, x_2, \ldots, x_m$ wrt an upper limit ℓ for x_1, where the feature x_1 bounds the size of the generated object, leading to a finite set of solutions. Let $I(\mathcal{O})$ be the instances of the combinatorial object. $M(\mathcal{C})$ is defined by:

$$M(\mathcal{C}) \iff \begin{cases} x_1 \leq \ell \quad (1) & (x_1, x_2, \ldots, x_m) \in I(\mathcal{O}) \quad (2) \\ C_1 \wedge C_2 \wedge \ldots \wedge C_{|\mathcal{C}|} \quad (3) & x_i \in \mathrm{dom}(x_i)\ \forall i \in \{1, \ldots, m\} \quad (4) \end{cases}$$

The constraint (1) bounds a feature (often chosen to be the size) to make the number of solutions finite. The constraint (2) forces the features to be consistent with the definition of the combinatorial object. That is, the features must have values that correspond to an instance of the combinatorial object, e.g. a graph could not have nine edges but only two nodes. This constraint (2) can be encoded according to the modeller's preferences. It can be seen as the modeller's prior knowledge of the constraint. This may be a decomposition into constraints that entirely define the possible values for the features. The constraint (3) is the conjunction of the conjectures in $\mathcal{C}$. Let Enumerate be an algorithm that takes

Algorithm 1: SelectOne($F, C, nback$)

```
1 if |C| = 1 then
2 |   nback' ← Enumerate(M(F));
3 |   if nback = nback' then return (∅, C) else return (C, ∅);
4 Evenly partition C into sets C1, C2; nback' ← Enumerate(M(F ∪ C2));
5 if nback = nback' then // K stands for Keep, R for Reject
6 |   (K, R) ← SelectOne(F, C2, nback); return (K, C1 ∪ R);
7 else return SelectOne(F ∪ C2, C1, nback) ;
```

Algorithm 2: Selection(C)

```
1 nback ← Enumerate(M(C)); F ← ∅;
2 repeat
3 |   (K, R) ← SelectOne(F, C, nback); F ← F ∪ K; C ← C \ (K ∪ R);
4 until K = ∅ ∨ C = ∅;
5 return F;
```

as input a model that returns the number of backtracks to enumerate all the solutions by a solver.

Alg. 1 takes as input a set of forced conjectures F, a set of candidate conjectures C, and the number of backtracks $nback$ taken by the solver to enumerate all solutions using all conjectures found by the Bound Seeker. It returns a tuple of two sets: A set K containing a single conjecture to keep, as it contributes to the filtering or an empty set if none exists, and a set R of conjectures to reject, as they do not contribute to any filtering. Alg. 1 works as a binary search to find the first constraint that impacts the filtering with $O(\log |C|)$ calls to the solver. If enumerating the solutions of $M(F \cup C_2)$ triggers $nback$ backtracks, then the constraints in C_1 do not filter values that are not already filtered by the constraints in $F \cup C_2$. We can reject the constraints C_1 and continue searching in C_2. Otherwise, the enumeration of $M(F \cup C_2)$ triggers more backtracks, and there exists at least one constraint in C_1 that captures a filtering missed by C_2. The binary search can continue in C_1. Alg. 2 calls Alg. 1 multiple times and extracts, one by one, the conjectures that have an impact on the filtering. It executes $O(|F| \log(|C|))$ calls to the solver where $|F|$ is the number of selected conjectures. The algorithms might return different sets of constraints depending on the order they eliminate the constraints. They might not find the smallest subset, but the subset is minimal, since removing any conjecture increases the number of backtracks. We also did an incremental version of Alg. 2 in which calls to the Enumerate function in Alg. 1 were optimised. These calls detect whether enumerating all solutions with a subset of conjectures takes as many backtracks as enumerating using all conjectures. The enumeration can stop as soon as the solver finds a solution with more backtracks than it required while using all constraints. In such a case, $nback \neq nback'$ even if we let the enumeration complete. Selected conjectures can be proved using a theorem prover or by a

human to become theorems. Adding the constraints to the model and letting the solver filter them leads to a correct and reinforced filtering.

4 Defining the Partition and the BinSeq Constraints

Humans are interested in balanced solutions for assignment problems. Examples of such problems are (i) problems where the total amount of work or time assigned to each worker or the amount of coursework to be validated by students for each session in an academic curriculum must be evenly distributed, and (ii) problems where the shift lengths assigned to each nurse should not vary too much, while meeting the demands of each period and the regulation. In this context, several constraints, such as NValue [24], Balance [7,11], Among [9] and Stretch [25] were introduced in constraint programming; these constraints are unified by the Partition and the BinSeq constraints.

4.1 Defining the Partition Constraint

A partition of a set U is a collection of subsets $S_1, S_2, \ldots, S_P$ that are pairwise disjoint, but whose union gives U. An array $\mathcal{X} = [X_1, X_2 \ldots, X_n]$ of integers encodes a partition as follows. The value $i \in U$ belongs to the subset S_j if and only if $X_i = j$. The partition has six features: (i) n the dimension of the array, i.e. the cardinality of U, (ii) P the number of distinct values in $\mathcal{X}$, i.e. the number of subsets, (iii) $\underline{M}$ (resp. (iv) $\overline{M}$) the number of occurrences of the least (resp. most) frequent integer in $\mathcal{X}$, (v) $\underline{\overline{M}}$ the difference $\overline{M} - \underline{M}$, (vi) S the sum of the squares of the number of occurrences for each distinct integer in $\mathcal{X}$.

Definition 1. *We define* Partition$([X_1, X_2, \ldots, X_n], P, \underline{M}, \overline{M}, \underline{\overline{M}}, S)$ *as a constraint satisfied if and only if*

$$P = |\{X_1, X_2, \ldots, X_n\}| \qquad S = \sum_{j \in \mathcal{X}} |\{i \mid X_i = j\}|^2 \tag{5}$$

$$\underline{M} = \min_{j \in \mathcal{X}} |\{i \mid X_i = j\}| \qquad \overline{M} = \max_{j \in \mathcal{X}} |\{i \mid X_i = j\}| \qquad \underline{\overline{M}} = \overline{M} - \underline{M} \tag{6}$$

Example 1 (Example for the Partition *Constraint).* Given the array $\mathcal{X} = [1, 5, 6, 1, 1, 1, 1, 1, 1, 1, 6]$, Partition$(\mathcal{X}, 3, 1, 8, 7, 69)$ holds, as $\mathcal{X}$ contains $P = 3$ distinct values, the least (resp. most) frequent one being value 5 (resp. 1) with $\underline{M} = 1$ (resp. $\overline{M} = 8$) occurrences, the difference between the occurrences of the most and the least frequent used values being $\underline{\overline{M}} = 7$, and the sum of the squares of the number of occurrences of distinct values being $S = 8^2 + 2^2 + 1^2 = 69$.

A Decomposition of the Partition *Constraint* Partition generalises NValue where $\mathcal{X}$ are the values, P is the number of values, and the remaining variables are ignored. As enforcing domain consistency on NValue [10] is NP-Hard, so is it for Partition. Therefore, we introduce the following decomposition named (`DECOMP_PART`). The Partition constraint can be encoded using the Global

Cardinality (GCC) and the NVALUE constraints as follows. Constraint (9) is redundant, but improves filtering. Let v be an upper bound on the number of partitions. For instance, one can fix v to $\max_{i\in[1,n]} \max(\mathrm{dom}(X_i))$.

$$\textsc{NValue}(\mathcal{X}, P) \quad (7) \qquad \textsc{GCC}(\mathcal{X}, [O_1, O_2, \ldots, O_v]) \quad (8)$$

$$\sum_{j=1}^{v} O_j = n \ \wedge \ \sum_{i=1}^{n} X_i = \sum_{j=1}^{v} j \cdot O_j \quad (9) \qquad \textsc{Min0}([O_1, O_2, \ldots, O_v], \underline{M}) \quad (10)$$

$$\overline{M} = \max(O_1, \ldots, O_v) \ \wedge \ \underline{\overline{M}} = \overline{M} - \underline{M} \ \wedge \ S = \sum_{j=1}^{v} O_j^2 \quad (11)$$

The $\textsc{Min0}([O_1, O_2, \ldots, O_v], \underline{M})$ constraint holds for the variables $O_1, \ldots, O_v$ with domains in $[0, n]$ iff at least one variable O_i (with $i \in [1, v]$) has a value in $\mathbb{N}^+$, and $\underline{M}$ is the smallest such value. Introducing the auxiliary variables $A_0, A_1, \ldots, A_v$, $\textsc{Min0}([O_1, O_2, \ldots, O_v], \underline{M})$ can be reformulated as (i) $A_0 = n$, (ii) $\forall i \in [1, v] : O_i = 0 \Rightarrow A_i = A_{i-1}$, $O_i > 0 \wedge O_i \geq A_{i-1} \Rightarrow A_i = A_{i-1}$, $O_i > 0 \wedge O_i < A_{i-1} \Rightarrow A_i = O_i$, and (iii) $\underline{M} = A_v \ \wedge \ \underline{M} > 0$.

4.2 Defining the BinSeq Constraint

Consider an array $\mathcal{X} = [X_1, X_2, \ldots, X_n]$ of 0/1 integers, where a *stretch* is a subsequence of maximal length of successive 1, and an *inter-distance* is a subsequence of maximum length of successive 0 located between 2 consecutive stretches.

Definition 2. *We define* $\textsc{BinSeq}([X_1, X_2, \ldots, X_n], N_1, G, \underline{G}, \overline{G}, \underline{\overline{G}}, GS, \underline{D}, \overline{D}, \underline{\overline{D}}, DS)$ *as a constraint satisfied if and only if*

- N_1 *is the number of values* 1 *in the sequence,*
- G *is the number of stretches of 1s,*
- $\underline{G}$ *(resp.* $\overline{G}$*) is the length of the smallest (resp. longest) stretch of 1s,*
- $\underline{\overline{G}}$ *is the difference between the lengths of the longest and the smallest stretch,*
- GS *is the sum of the squared lengths of the stretches of 1s,*
- $\underline{D}$ *(resp.* $\overline{D}$*) is the length of the smallest (resp. longest) inter-distance of 0s,*
- $\underline{\overline{D}}$ *is the difference* $\overline{D} - \underline{D}$*,*
- DS *is the sum of the squared lengths of the inter-distances of 0s.*

When there is no stretch, $\underline{G} = \overline{G} = 0$*; when there is no inter-distance,* $\underline{D} = \overline{D} = 0$.

Example 2 (Example for the BinSeq *Constraint).* Given the array $\mathcal{X} = [0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1]$, the $\textsc{BinSeq}(\mathcal{X}, 7, 3, 1, 4, 3, 21, 1, 2, 1, 5)$ constraint holds, as the array $\mathcal{X}$ contains $N_1 = 7$ occurrences of 1s, $G = 3$ stretches of minimum (resp. maximum) length $\underline{G} = 1$ (resp. $\overline{G} = 4$) with $\underline{\overline{G}} = \overline{G} - \underline{G} = 3$, the sum of the squared lengths of the stretches $GS = 2^2 + 4^2 + 1^2 = 21$, the minimum (resp. maximum) inter-distance $\underline{D} = 1$ (resp. $\overline{D} = 2$) with $\underline{\overline{D}} = \overline{D} - \underline{D} = 1$, and the sum of the squared lengths of the inter-distances $DS = 2^2 + 1^2 = 5$.

Decomposing the BinSeq *Constraint* For any $M \in \{N_1, G, \underline{G}, \overline{G}, \overline{\underline{G}}, GS, \underline{D}, \overline{D}, \overline{\underline{D}}\}$, an automaton with $O(n)$ states accepts the sequence $X_1, X_2, \ldots, X_n, M$. For $M = DS$, the automaton has $O(n^2)$ states. The conjunction of these automata has $O(n^{11})$ states, proving that domain consistency can be enforced on BinSeq in polynomial time using the Regular constraint. Due to such size, we created two decompositions of the BinSeq constraint for use in our experiments, which we describe briefly for space reasons. The first decomposition, named (`DECOMP_SEQ_GCC`), has the advantage of using standard constraints and can therefore be encoded in MiniZinc. The key idea is to associate a unique odd (or even) number with each stretch (or inter-distance) in the array $\mathcal{X}$. For instance, the array $\mathcal{X}$ = [0,0,0,1,1,0,0,1,1,1,1,0,1] from Example 2 is mapped to $\mathcal{X}' = [0, 0, 0, 1, 1, 2, 2, 3, 3, 3, 3, 4, 5]$. Then a Global Cardinality Constraint (GCC) on the array $\mathcal{X}'$ exposes the number of occurrences of each value in $\mathcal{X}'$. The occurrence variables of this GCC constraint can easily express the ten features of the BinSeq constraint: occurrences of even (resp. odd) values correspond to the lengths of stretches of 0s (resp. 1s). The 2nd decomposition, named (`DECOMP_SEQ_AUT`), is based on the SICStus register automaton constraint [6], which is not available in MiniZinc. For each feature f in the BinSeq constraint, we associate a register automaton with at most 5 states to link the array $\mathcal{X}$ to the feature f. When tested with SICStus, this 2nd decomposition proved to filter the BinSeq constraint variables much better than the first decomposition.

4.3 Challenge Behind the Partition and the BinSeq Constraints

The weakness of the decompositions (`DECOMP_PART`), (`DECOMP_SEQ_GCC`) and (`DECOMP_SEQ_AUT`) of the Partition and BinSeq constraints is that while most of the features of these two constraints are linked to the variables $X_1, X_2, \ldots, X_n$, there is virtually no direct link between the features, apart from the equalities $\overline{\underline{M}} = \overline{M} - \underline{M}$, $\overline{\underline{G}} = \overline{G} - \underline{G}$ and $\overline{\underline{D}} = \overline{D} - \underline{D}$. As these features do not vary independently, this poses a challenge for getting an efficient filtering algorithm. To alleviate this problem, we show in Sect. 5 how to extract such missing links.

5 A Machine Assisted Generated Filtering Algorithm with an Application to Partition and BinSeq

We use the framework presented in Sect. 3 to strengthen the decompositions provided for Partition and BinSeq. We apply Alg. 2 to select, from a set of conjectures generated by the Bound Seeker for the partition and 0–1 sequence combinatorial objects, those that do not reduce the amount of filtering when all conjectures are used. We show how we generate the conjectures, how we use Alg. 2 to select them, and we prove the most complicated selected conjecture.

Generating the Conjectures for Partition *and* BinSeq The first step is to generate the conjectures using the Bound Seeker. We provided instances of partitions

and 0–1 sequences $I(\mathcal{O})$ of size $n \leq 30$ and reused CP models, which break symmetries where the different parts of a partition and the different stretch lengths are sorted by increasing size. For each object, we use all the features introduced in the PARTITION and BINSEQ constraints as *primary features*. For the PARTITION constraint, we also manually introduced a set of *secondary features* at the partition object level, which the Bound Seeker uses to systematically search for all the sharp bounds of the partition object's features, whereas for the BINSEQ constraint *we did not introduce any secondary features to avoid any human assistance*. These instances of partitions (or 0–1 sequences) were sent to the Bound Seeker, which returned 92 (or 3739) conjectures.

Generating Secondary Features for PARTITION Most of these secondary features correspond to conditional expressions, where the value associated with the base case is either the constant 0, or is tied only to the size of the smallest partition. As the primary features focus on the largest and smallest partitions, the secondary features provide statistics on the other partitions. VV is the number of partitions, excluding the largest and smallest, and NN is the number of values that are not in the smallest or largest partition. A is the average size of the partitions that are neither the smallest nor the largest, rounded down. $A = 0$ if no such partition exists. SS is the sum of the squared sizes of the smallest and largest partitions. If there is only one partition, SS is its squared size.

$$VV = \begin{cases} P - 2 \text{ if } P > 1 \\ 0 \quad \text{otherwise} \end{cases} \quad (12) \qquad NN = \begin{cases} n - \underline{M} - \overline{M} \text{ if } P > 1 \\ 0 \quad \text{otherwise} \end{cases} \quad (13)$$

$$A = \begin{cases} \lfloor \frac{NN}{VV} \rfloor \text{ if } P > 2 \\ 0 \quad \text{otherwise} \end{cases} \quad (14) \qquad SS = \begin{cases} \underline{M}^2 + \overline{M}^2 \text{ if } P > 1 \\ \underline{M}^2 \quad \text{otherwise} \end{cases} \quad (15)$$

R is the number of elements remaining after removing $\underline{M}$ elements from each partition. MID is, when the number of largest partitions is maximal, the size of an intermediate partition, either the one in between the smallest and largest, or the smallest, if no such partition exists. RR is the number of largest partitions or 0 if all partitions have the same size. SM is the gap between the squared sizes of the smallest and largest partitions. $SMIN$ is the sum of the squared sizes of partitions when they are all at the smallest size and with one partition excluded.

$$MID = \begin{cases} \underline{M} + (R \bmod \overline{M}) \text{ if } \overline{M} > 0 \\ \underline{M} \quad \text{otherwise} \end{cases} \quad (16) \qquad R = n - P \cdot \underline{M} \quad (17)$$

$$RR = \begin{cases} \left\lfloor \frac{R}{\overline{M}} \right\rfloor \text{ if } \overline{M} > 0 \\ 0 \quad \text{otherwise} \end{cases} \quad (18) \qquad SM = \overline{M}^2 - \underline{M}^2 \quad (19)$$

$$SMIN = \underline{M}^2 \cdot (P - 1) \quad (20)$$

Applying Alg. 2 to PARTITION *and* BINSEQ We implemented Alg. 2 in SICStus using the clp(FD) solver on a Mac Studio M2 Ultra. We define $M(\mathcal{C})$ as the decomposition (`DECOMP_PART`) for PARTITION and (`DECOMP_SEQ_AUT`) for BINSEQ. To limit the number of conjectures to be proved, we impose a limit of 20 on the number of conjectures returned by Alg. 2, and run the selection algorithm on increasing value of n. We also stopped the search process when the same set of conjectures was selected for three consecutive values of n. Alg. 2 selects 4 conjectures out of 92 with $n = 7$ for PARTITION in 0.5 sec and 17 conjectures out of 3728 with $n = 10$ for BINSEQ in 150 min. Using an incremental version of the selection algorithm sketches at the end of Sect. 3.1, these two times were reduced to 0.1 sec and 17 min. We have proved all the selected conjectures listed in Table 1, and we now focus on the most interesting one for the BACP problem, which is a sharp lower bound on the sum of the squares of the partition sizes wrt the n, P, $\underline{M}$ and $\overline{M}$ features of the PARTITION constraint.

A Sharp Lower Bound for the Sum of the Squares of the Partition Sizes Using the four secondary features introduced in (12)–(15), the Bound Seeker returned the following conjecture for which we provide a proof and intuition.

$$S \geq -A^2 \cdot VV - A \cdot VV + 2 \cdot A \cdot NN + SS + NN \tag{21}$$

The intuition behind (21) is that, to achieve the smallest possible sum of squares, the largest partition has size $\overline{M}$, the smallest partition has size $\underline{M}$, and the other partitions have size NN/VV. But since the size must be an integer, NN mod VV partitions have their size rounded up to $A+1$, while the rest have their size rounded down to A. To prove (21), we first need to introduce Theo. 1.

Theorem 1 (minimisation of $S = \sum_i^P y_i^2$). *Let $y_1, y_2, \cdots, y_P$ be positive integers whose sum is equal to n and which minimise $S = \sum_{i=1}^P y_i^2$. Then n mod P of these integers are equal to $\lfloor n/P \rfloor + 1$, and the others are all equal to $\lfloor n/P \rfloor$.*

Proof. $y_1, y_2, \cdots, y_P$ minimise $\sum_{i=1}^P y_i^2$ only if $y_i - y_j \leq 1$. Assume i and j with $y_i - y_j \geq 2$ then $(y_i - 1)^2 + (y_j + 1)^2 + \sum_{k \notin \{i,j\}}^P y_k^2 = \sum_k^P y_k^2 - 2(y_i - y_j) + 2 < y_i^2 + y_j^2 + \sum_{k \notin \{i,j\}}^P y_k^2 = \sum_k^P y_k^2$. As $(y_i - 1) + (y_j + 1) + \sum_{k \notin \{i,j\}}^P y_k = n$, the previous strict inequality shows that the decomposition $y_1 - 1, y_2 + 1, y_3 \cdots, y_P$ minimises better $\sum_{i=1}^P y_i^2$ than the decomposition $y_1, y_2, \cdots, y_P$, a contradiction. The only way to have $y_i - y_j \leq 1$ and $\sum_i^P y_i = n$ is to set the values of the integers, as stated by Theo. 1, otherwise, we always have $\sum_{k=1}^P y_k \neq n$. □

Proof (Conjecture (21)*).* If $P = 1$, substituting (12) to (15) into (21) simplifies to $S \geq n^2$ which is tight and consistent with the definition PARTITION. If $P = 2$, we have $n = \overline{M} + \underline{M}$. By substituting (12), Conj. (21) is simplified by $S \geq \overline{M}^2 + \underline{M}^2$, which is also tight and consistent. If $P > 2$, then $A = \lfloor \frac{NN}{VV} \rfloor$ leading to $NN = A \cdot VV + NN$ mod VV. After substituting NN by $A \cdot VV + NN$ mod VV inside Conj. (21) we have $S \geq -A^2 \cdot VV + 2 \cdot A^2 \cdot VV + 2 \cdot A \cdot (NN \text{ mod } VV) + SS +$

Table 1. Bounds selected by Alg. 2 for (A) PARTITION and (B) BINSEQ

(A) $\overline{M} \leq \min(P \cdot \overline{M} - n, \overline{M} - 1)$ $\quad \underline{\overline{M}} \leq n - P \cdot \underline{M}$

$S \geq -A^2 \cdot VV - A \cdot VV + 2 \cdot A \cdot NN + SS + NN$ (21) $\quad S \leq MID^2 + SM \cdot RR + SMIN$

(B) $N_I \leq \min(G \cdot \overline{G}, n - G + 1)$ $\quad \overline{G} \leq \begin{cases} n + \underline{\overline{G}} & \text{if } \underline{\overline{G}} = n \cdot \underline{\overline{D}} \\ \left\lfloor \frac{n - \underline{\overline{G}} - \underline{\overline{D}} - \min(\underline{\overline{D}}, 1) - 1}{\min(\underline{\overline{D}}, 1) + 2} \right\rfloor + \underline{\overline{G}} & \text{otherwise} \end{cases}$

$\overline{G} \geq \left\lfloor \frac{n}{n - N_I + 1} \right\rfloor$ $\quad \overline{D} \leq [\![G \geq 2]\!] \cdot (n - G \cdot \underline{G} - G + 2)$ $\quad GS \geq \underline{G}^2 \cdot G$

$\underline{D} \leq \begin{cases} 0 & \text{if } G \leq 1 \\ \left\lfloor \frac{n - \overline{G} + 1 - G}{G - 1} \right\rfloor & \text{if } G > 1 \end{cases}$ $\quad \underline{\overline{G}} \leq \begin{cases} n & \text{if } G = 1 \wedge \overline{D} = 0 \\ \min(G, 1) & \text{if } G \neq 1 \wedge \overline{D} = 0 \\ n - \overline{D} - (G-2) \cdot \underline{D} - G + \min(G, 1) & \text{if } G \neq 1 \wedge \overline{D} \geq 1 \end{cases}$

$GS \geq \max(\overline{G}^2 + 1 - [\underline{D} = 0] - [\overline{G} = 0], 0)$ $\quad GS \leq \begin{cases} \max({N_I}^2 + G - 1, 0) & \text{if } G \leq 1 \\ \max((N_I - G + 1)^2 + G - 1, 0) & \text{otherwise} \end{cases}$

$GS \leq \begin{cases} \max({N_I}^2, 0) & \text{if } \underline{\overline{D}} = 0 \wedge \min(N_I, 1) = 1 \\ 0 & \text{if } \underline{\overline{D}} = 0 \wedge \min(N_I, 1) = 0 \\ \max((N_I - 2)^2 + 2, 0) & \text{if } \underline{\overline{D}} \geq 1 \end{cases}$ $\quad DS \geq \underline{D}^2 \cdot (G - 1)$

$DS \geq \overline{D}^2$ $\quad DS \leq \begin{cases} 0 & \text{if } N_I \leq 1 \\ (n - N_I)^2 & \text{otherwise} \end{cases}$ $\quad DS \geq \begin{cases} 0 & \text{if } G \leq 1 \\ \max((\underline{\overline{D}} + 1)^2 + G - 2, 0) & \text{otherwise} \end{cases}$

$DS \leq \begin{cases} \max((n - N_I - (G-2))^2 + G - 2, 0) & \text{if } G \geq 2 \\ \max(G - 2, 0) & \text{otherwise} \end{cases}$ $\quad GS \geq \underline{\overline{G}} \cdot (\underline{\overline{G}} + 1) \cdot \min(G, 1) + \underline{\overline{G}} + G$

$GS \leq \begin{cases} \max(n^2, 0) & \text{if } G = 1 \wedge \overline{D} = 0 \\ \max((\min(G, 1))^2 + G - 1, 0) & \text{if } G \neq 1 \wedge \overline{D} = 0 \\ \max((n - \overline{D} - (G-2) \cdot \underline{D} - G + 1)^2 + G - 1, 0) & \text{if } G \neq 1 \wedge \overline{D} \geq 1 \end{cases}$

$NN \bmod VV$ which simplifies to $S \geq A^2 \cdot VV + (2 \cdot A + 1) \cdot (NN \bmod VV) + SS$. By substituting $(2 \cdot A + 1)$ by $(A+1)^2 - A^2$, and SS by $\overline{M}^2 + \underline{M}^2$, we obtain this inequality that is proved by Theo. 1: $S \geq \overline{M}^2 + \underline{M}^2 + (A+1)^2 \cdot (NN \bmod VV) + A^2 \cdot (VV - (NN \bmod VV))$ □

6 Experiments on the BACP and the BSSP Problems

We test our framework on the Balanced Academic Curriculum Problem (BACP) [22] and the Balanced Shift-Scheduling Problem [15]. These problems involve

combinatorial objects, such as partitions or binary sequences, where multiple features are restricted, making sharp bounds essential. We describe the BACP and the two models we use for it. We outline the BSSP and the two models we developed for it. We evaluate these models with and without using the conjectures selected by Alg. 2 on the Chuffed [19] and SICStus clp(FD) [13] solvers.

6.1 Describing the BACP Problem and its Models

Problem Description The BACP is a problem in which a set of n courses must be assigned to v periods. Let X_i denote the period of course i. A lower and upper limit $\underline{m}$ and limit $\overline{m}$ for the number of courses per period must be respected. Let $\mathcal{P}rec$ be the set of pairs of courses (i, k) such that i is the prerequisite for k, and let $c(i)$ be the number of credits for course i. The *load*, i.e. the sum of the credits of the assigned courses, for each period, must be in $[\underline{L}_0, \overline{L}_0]$. The course load must be balanced over the different periods. There are many ways to balance the load. One can minimise the maximum load or the load range. To distribute the load more evenly, we minimise the sum of the squares of the loads.

Description of the Models The first model, which we call (`BACP_PART`), contains the constraints (22) to (30). The array of variables $[X_1, X_2, \ldots, X_n]$ partitions the courses $1 \ldots n$ into time periods $1 \ldots v$. The constraints (23), (25), (26), and (27) enforce a lower and upper limit on the number of courses taught per period. The constraint (28) enforces precedences between the corresponding courses i and k. The PARTITION constraint (31) imposes that the parameter S is the sum of the squares for the load of each time period. The array $[\underbrace{X_1, \ldots,\ X_1}_{c(1)\text{ times}}, \ldots, \underbrace{X_n, \ldots,\ X_n}_{c(n)\text{ times}}]$ contains $c(i)$ occurrences of variable X_i for each course i. One occurrence of a value of X_i represents one credit of the course i. As X_i takes a value of a time period, the number of occurrences of a time period j in the array is the sum of credits of all courses for that time period, i.e. the load for the period j is $\sum_{i=1}^{n} [X_i = j] \cdot c(i)$. Constraint (31) also ensures that $\overline{L}$ is the range between the minimal and maximal load $\underline{L}$ and $\overline{L}$. Constraint (29) ensures the respect of the lower and upper limits of the load per period. (30) define the initial domains of the variables X_i and ensure that the time periods that are used are contiguous. The model minimises S for a better balance of load among periods, rather than minimising $\overline{L}$ or $\overline{L}$, as it is usually done.

$$\text{Minimize } S \tag{22}$$

$$\text{GCC}([X_1, X_2, \ldots, X_n], [M_1, \ldots, M_v]) \tag{23}$$

$$\sum_{j=1}^{v} M_j = n \wedge \sum_{i=1}^{n} X_i = \sum_{j=1}^{v} j \cdot M_j \tag{24}$$

$$\underline{M} = \min(M_1, \ldots, M_v) \tag{25}$$

$$\overline{M} = \max(M_1, \ldots, M_v) \tag{26}$$

$$\underline{m} \leq \underline{M} \ \wedge\ \overline{M} \leq \overline{m} \tag{27}$$

$$\forall (i, k) \in \mathcal{P}rec, X_i < X_k \tag{28}$$

$$\underline{L}_0 \leq \underline{L} \quad \wedge\ \overline{L} \leq \overline{L}_0 \tag{29}$$

$$\forall i \in [1,n], 1 \leq X_i \leq v \wedge X_i \leq P \tag{30}$$

$$\textsc{Partition}([\underbrace{X_1,\ldots,\ X_1}_{c(1)\text{ times}},\ldots,\underbrace{X_n,\ldots,\ X_n}_{c(n)\text{ times}}], P, \underline{L}, \overline{L}, \underline{\overline{L}}, S) \tag{31}$$

A second model, we call (`BACP_CUM`), by Mats Carlsson [22], uses constraints (22) to (28) and (30) and encodes the rest using a Cumulative constraint.

6.2 Describing the BSSP Problem and its Models

Problem Description: We present the Balanced Shift-Scheduling Problem (BSSP) to optimise employee schedules, balancing work and rest periods, with no requirement for equal number of rests per employee. There is only one activity on which all employees work on. An employee has lunch and has rests between periods of work. An employee rests for at least four hr at the beginning and end of a day, with work periods ranging from 1 to 4 hr between rests. We divide a day into 96 slots of 15 mins. When the rest is between periods of work, it is at least 15 mins. long and not more than an hr long. Lunch is one hr long. An employee should have at least three rest periods during the workday, which should be between 6 and 8 hr. At time i, there should be m_i employees working. Let M_i be the actual number of employees working at time i. There is a penalty cost $\overline{p}_i$ associated with overemployment and a penalty cost $\underline{p}_i$ associated with underemployment. The total penalty cost is $\sum_{i=1}^{96}(M_i > m_i) \cdot \overline{p}_i + (M_i < m_i) \cdot \underline{p}_i$. As we seek the most balanced work-rest distribution, we add to that penalty cost the sum of squared work periods GS_e and rest periods DS_e for each employee e.

Models Description: We encode an employee schedule with an array $\mathcal{X} = [X_1, \ldots, X_n]$ of 0/1 integers, where 1 represents a 15-min. work period and 0 a 15-min. rest period; e.g., [000111111111111001111000011110111101111011111000] means that the employee starts with a 45-min. rest, works for 3 hr, takes a 30-min. break, and then works 1 hr before lunch. After lunch, s/he finishes his day with 4 hr of work, separated by 15-min. breaks. We are interested by the following features of $\mathcal{X}$: its size n, G the number of stretches of 1, i.e. the number of periods of work, N_1 the number of 1 in the array, i.e. the total work time, $\underline{G}$ ($\overline{G}$) the smallest (largest) length of a stretch of 1, i.e. the shortest (longest) period of work, $\underline{\overline{G}}$ the difference $\overline{G} - \underline{G}$, GS the sum of the squares of the stretch lengths, $\underline{D}$ ($\overline{D}$) the smallest (largest) inter-distance between consecutive stretches of 1, i.e. the smallest (largest) rest between work periods, $\underline{\overline{D}}$ the difference $\overline{D} - \underline{D}$, DS the sum of the squares of the inter-distance lengths.

We encode the BSSP using the BinSeq constraint. Let m be the maximum number of employees required in a day. Constraint (32) ensures balanced schedules and the lowest penalty cost. (33) connects the decision variables $\mathcal{X}_e = [X_{1e}, \ldots, X_{ne}]$ to $n, N_1, G, \underline{G}, \overline{G}, \underline{\overline{G}}, GS, \underline{D}, \overline{D}, \underline{\overline{D}}, DS$. The variable X_{ie} is 1 iff employee e works at time i. (34) enforces employees to take at least 4 breaks of at least 15 min. and no more than 1 hr, including 1 lunch break of exactly 1

hr. (35) catches the number of employees working in each time slot. (36) ensures that an employee works between 6 and 8 hr and between 1 hr and 4 hr before a rest. As employees are required to have at least 4 hr of rest at the beginning and end of each workday, we exclude the first and last 16 time slots.

$$\text{Minimize } \sum_{e=1}^{m} GS_e + DS_e + \sum_{i=1}^{96} (M_i > m_i) \cdot \overline{p}_i + (M_i < m_i) \cdot \underline{p}_i \quad (32)$$

$$\forall e \in [1:m], \textsc{BinSeq}(\mathcal{X}_e, N_{1\,e}, G_e, \underline{G}_e, \overline{G}_e, \underline{\overline{G}}_e, GS_e, \underline{D}_e, \overline{D}_e, \underline{\overline{D}}_e, DS_e) \quad (33)$$

$$1 \leq \underline{D}_e \wedge \overline{D}_e = 4 \wedge 4 \leq G_e \quad (34) \qquad \forall i \in [1:64], M_i = \sum_{e=1}^{m} X_{ie} \quad (35)$$

$$24 \leq N_{1\,e} \leq 32 \ \wedge 4 \leq \underline{G}_e \wedge \overline{G}_e \leq 16 \quad (36)$$

Using the (`DECOMP_SEQ_GCC`) and (`DECOMP_SEQ_AUT`) decompositions of the BinSeq constraint, we obtain the two models (`BSSP_GCC`) and (`BSSP_AUT`).

6.3 Evaluation

Settings for the BACP We evaluate the models on 31 real-world instances with up to 10 periods and 42–66 courses, all from CSPLib [1] with SICStus and Chuffed. For SICStus, we use two strategies: (*i*) When the learnt bounds are unused, the 1st strategy selects the most constrained variable from $\mathcal{X} = X_1, \ldots, X_n$, and performs a dichotomic search. (*ii*) The 2nd strategy branches first on $\underline{\overline{L}}, S, P, \underline{L}, \overline{L}$, then on $\mathcal{X}$ using the 1st strategy, prioritising feature variables that makes the objective smaller. As it is more aggressive, we use it only with models that include bounds. For Chuffed, using the free search, the search branches on $\underline{\overline{L}}, S, P, \underline{L}, \overline{L}$, and chooses the values in ascending order. Then, it branches on $\mathcal{X}$ in that order and chooses the values in descending order.

Settings for the BSSP We evaluate the models on 10 real instances with 2–7 employees. For SICStus, we use an aggressive strategy that, after restricting $\overline{G}$, fixes the variables by increasing time slot i, restricting first the covering cost $(M_i > m_i) \cdot \overline{p}_i + (M_i < m_i) \cdot \underline{p}_i$, and then the variables X_{ie} for time slot i. For Chuffed in free search, we enumerate on variables X_{ie} by increasing time slot i.

Results We generate the implied constraints only once, not for each instance. This process is roughly equivalent to a human designing and programming a filtering algorithm. It took the Bound Seeker about one week on the Digital Research of Alliance of Canada clusters to generate all conjectures. We used a single core of a Mac Studio M2 Ultra with a fifteen-min. timeout. For each model and solver, we report the number of instances where the solver (*i*) proves optimality (#optimality), (*ii*) the timeout (#timeout), (*iii*) finds no solution (#not found), and (*iv*) the number of instances for which a model is the best (#best) for the

Table 2. Results of the experiments where the best results are shown in bold

Problem	BACP								BSSP			
Solver	SICStus				Chuffed				SICStus		Chuffed	
Model	(BACP_PART)		(BACP_CUM)		(BACP_PART)		(BACP_CUM)		(BSSP_AUT)		(BSSP_GCC)	
Used bounds	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
#optimality	0	12	0	30	0	**31**	0	30	0	0	**6**	**6**
#timeout	31	17	31	1	31	**0**	31	1	10	10	**4**	**4**
#not found	2	19	2	**0**	**0**	**0**	**0**	**0**	5	**0**	**0**	**0**
#best	0	0	0	**19**	0	10	0	2	0	3	1	**6**
$\sum$obj	/	/	/	**222359**	225375	**222359**	232769	222529	/	5697	**5524**	7005592
$\sum$time	/	/	/	17min	5h	**5min**	5h	29min	/	137min	81min	**71min**

corresponding solver. The best models are those proving optimality faster. If none do, we select the one with the lowest objective value in the least time. We also provide the sum of objective values ($\sum$obj) and the sum of times spent ($\sum$time) to find the best solution for all instances where a solution was found. Table 2 gives the aggregated results. For the BACP, none of the models proves optimality without using the bounds. When using bounds, although (BACP_PART) proves optimality for 12 instances with SICStus, it proves optimality for all instances with Chuffed.[1] The fastest model is (BACP_PART) with 222359 as the smallest sum of objective values found in 5min for all the instances when using bounds with Chuffed.[2] For the BSSP, Table 2 shows that the models find better solutions and are faster when using bounds. The fastest model is (BSSP_GCC) with 71 min as the total time on the 10 instances when using bounds, meaning that when using bounds, it is 1.14 times faster than the same model without using bounds. The sum of objective values 7005592 is large due to one instance where using bounds degraded the results. But for the other instances, the (BSSP_GCC) is six times the best among all models when using bounds. Although (BSSP_AUT) never proved optimality, using bounds, it found the best solution three times, but failed to find any solution for five instances without exploiting bounds.

7 Conclusion

Bound Seeker-generated conjectures can lead to better filtering for problems with multiple features of a combinatorial object. We synthesise redundant constraints supported by various solvers. Our algorithm addresses two major issues: the impossibility of proving thousands of conjectures produced by the Bound Seeker and the need to identify impactful conjectures to improve filtering.

[1] The 19 instances where SICStus with (BACP_PART) did not find any solution when using bounds is due to the aggressive search strategy quoted earliest.

[2] The incremental version of Alg. 2 and all conjecture proofs are in an arXiv report.

References

1. CSPLib: A problem library for constraints. https://www.csplib.org/, accessed: 2024-12-06
2. Achterberg, T., Bixby, R.E., Gu, Z., Rothberg, E., Weninger, D.: Presolve reductions in mixed integer programming. INFORMS J. Comput. **32**(2), 473–506 (2020). https://doi.org/10.1287/IJOC.2018.0857, https://doi.org/10.1287/ijoc.2018.0857
3. Aouchiche, M., Caporossi, G., Hansen, P., Laffay, M.: Autographix: a survey. Electron. Notes Discret. Math. **22**, 515–520 (2005). https://doi.org/10.1016/j.endm.2005.06.090, https://doi.org/10.1016/j.endm.2005.06.090
4. Arafailova, E., Beldiceanu, N., Simonis, H.: Deriving generic bounds for time-series constraints based on regular expressions characteristics. Constraints An Int. J. **23**(1), 44–86 (2018). https://doi.org/10.1007/s10601-017-9276-z, https://doi.org/10.1007/s10601-017-9276-z
5. Baptiste, P., Pape, C.L., Nuijten, W.: Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems. Kluwer (2012)
6. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In: Wallace, M. (ed.) Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3258, pp. 107–122. Springer (2004). https://doi.org/10.1007/978-3-540-30201-8_11, https://doi.org/10.1007/978-3-540-30201-8_11
7. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global constraint catalog, (revision a) (2012)
8. Beldiceanu, N., Cheukam-Ngouonou, J., Douence, R., Gindullin, R., Quimper, C.: Acquiring maps of interrelated conjectures on sharp bounds. In: Solnon, C. (ed.) 28th International Conference on Principles and Practice of Constraint Programming, CP 2022, July 31 to August 8, 2022, Haifa, Israel. LIPIcs, vol. 235, pp. 6:1–6:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.CP.2022.6, https://doi.org/10.4230/LIPIcs.CP.2022.6
9. Beldiceanu, N., Évelyne Contejean: Introducing global constraints in chip. Mathematical and Computer Modelling **20**(12), 97–123 (1994). https://doi.org/https://doi.org/10.1016/0895-7177(94)90127-9, https://www.sciencedirect.com/science/article/pii/0895717794901279
10. Bessiere, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: Filtering algorithms for the nvalue constraint. In: Barták, R., Milano, M. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Second International Conference, CPAIOR 2005, Prague, Czech Republic, May 30 - June 1, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3524, pp. 79–93. Springer (2005). https://doi.org/10.1007/11493853_8, https://doi.org/10.1007/11493853_8
11. Bessière, C., Hebrard, E., Katsirelos, G., Kızıltan, Z., Picard-Cantin, É., Quimper, C., Walsh, T.: The balance constraint family. In: O'Sullivan, B. (ed.) Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8656, pp. 174–189. Springer (2014). https://doi.org/10.1007/978-3-319-10428-7_15, https://doi.org/10.1007/978-3-319-10428-7_15
12. Biere, A., Järvisalo, M., Kiesl, B.: Handbook of Satisfiability, chap. 9: Preprocessing in SAT Solving, pp. 391–435. IOS Press (2021)

13. Carlsson, M., Mildner, P.: Sicstus prolog – the first 25 years. CoRR **abs/1011.5640** (2010), http://arxiv.org/abs/1011.5640
14. Charnley, J.W., Colton, S., Miguel, I.: Automatic generation of implied constraints. In: Brewka, G., Coradeschi, S., Perini, A., Traverso, P. (eds.) ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings. Frontiers in Artificial Intelligence and Applications, vol. 141, pp. 73–77. IOS Press (2006), http://www.booksonline.iospress.nl/Content/View.aspx?piid=1649
15. Demassey, S., Pesant, G., Rousseau, L.M.: A cost-regular based hybrid column generation approach. Constraints **11**(4), 315–333 (2006). https://doi.org/10.1007/s10601-006-9003-7, https://publications.polymtl.ca/23185/, aRRAY(0x55680943b278)
16. Fajtlowicz, S.: On conjectures of Graffiti. Discret. Math. **72**(1-3), 113–118 (1988). https://doi.org/10.1016/0012-365X(88)90199-9, https://doi.org/10.1016/0012-365X(88)90199-9
17. Gent, I.P., Jefferson, C., Linton, S., Miguel, I., Nightingale, P.: Generating custom propagators for arbitrary constraints. Artif. Intell. **211**, 1–33 (2014). https://doi.org/10.1016/J.ARTINT.2014.03.001, https://doi.org/10.1016/j.artint.2014.03.001
18. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Generating special-purpose stateless propagators for arbitrary constraints. In: Cohen, D. (ed.) Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6308, pp. 206–220. Springer (2010). https://doi.org/10.1007/978-3-642-15396-9_19, https://doi.org/10.1007/978-3-642-15396-9_19
19. Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, Kathryn Francis: Chuffed 0.12.1, a lazy clause generation solver. https://github.com/chuffed/chuffed (2023)
20. Gindullin, R., Beldiceanu, N., Ngouonou, J.C., Douence, R., Quimper, C.G.: Boolean-arithmetic equations: Acquisition and uses. In: Proceedings of the 20th International Conference on the Integration of Constraint Programming (CPAIOR) (2023)
21. Hansen, P., Caporossi, G.: Autographix: An automated system for finding conjectures in graph theory. Electron. Notes Discret. Math. **5**, 158–161 (2000). https://doi.org/10.1016/S1571-0653(05)80151-9, https://doi.org/10.1016/S1571-0653(05)80151-9
22. Hnich, B., Kiziltan, Z., Walsh, T.: CSPLib problem 030: Balanced academic curriculum problem (bacp). http://www.csplib.org/Problems/prob030 (1999)
23. Laurière, J.: Constraint propagation or automatic programming. Tech. Rep. 19, IBP-Laforia (1996), available at https://www.lri.fr/~sebag/Slides/Lauriere/Rabbit.pdf
24. Pachet, F., Roy, P.: Automatic generation of music programs. In: Jaffar, J. (ed.) Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1713, pp. 331–345. Springer (1999). https://doi.org/10.1007/978-3-540-48085-3_24, https://doi.org/10.1007/978-3-540-48085-3_24

25. Pesant, G.: A filtering algorithm for the stretch constraint. In: Walsh, T. (ed.) Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2239, pp. 183–195. Springer (2001). https://doi.org/10.1007/3-540-45578-7_13, https://doi.org/10.1007/3-540-45578-7_13
26. Pesant, G., Régin, J.C.: Spread: A balancing constraint based on statistics. In: Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP 2005). pp. 460–474 (2005)
27. Pierre Schaus, Yves Deville, P.D.: Bound-consistent deviation constraint. In: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007). pp. 620–634 (2007)
28. Rendl, A., Miguel, I., Gent, I.P., Gregory, P.: Common subexpressions in constraint models of planning problems. In: Bulitko, V., Beck, J.C. (eds.) Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA 2009, Lake Arrowhead, California, USA, 8-10 August 2009. AAAI (2009), http://www.aaai.org/ocs/index.php/SARA/SARA09/paper/view/823
29. Régin, J.C.: Modélisation et Contraintes Globales en Programmation par Contraintes. HDR dissertation, Université de Nice (2004)
30. Schaus, P., Deville, Y., Dupont, P., Régin, J.C.: The deviation constraint. In: Proceedings of the 4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2007). pp. 260–274 (2007)
31. Schaus, P., Deville, Y., Dupout, P., Régin, J.C.: Simplification and extension of the spread constraint. In: Proceedings of the third international workshop on constraint propagation and implementation. pp. 77–91 (2006)