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

Improved Filtering for the Bin-Packing with Cardinality Constraint

Guillaume Derval · Jean-Charles Régin · Pierre Schaus

Received: date / Accepted: date

Abstract Previous research shows that a cardinality reasoning can improve the pruning of the bin-packing constraint. We first introduce a new algorithm, called BPCFlow, that filters both load and cardinality bounds on the bins, using a flow reasoning similar to the Global Cardinality Constraint. Moreover, we detect impossible assignments of items by combining the load and cardinality of the bins, using a method to detect items that are either ”too-big” or ”too-small”. This method is adapted to two previously existing filtering techniques along with BPCFlow, creating three new propagators. We then experiment the four new algorithms on Balanced Academic Curriculum Problem and Tank Allocation Problem instances. BPCFlow is shown to be stronger than previously existing filtering, and more computationally intensive. We show that the new filtering is useful on a small number of hard instances, while being too expensive for general use. Our results show that the introduced ”too-big/too-small” filtering can most of the time drastically reduce the size of the search tree and the computation time. This method is profitable in 88% of the tested instances.

Keywords Bin-packing, cardinality, flows, constraints

G. Derval
UCLouvain, Belgium
E-mail: guillaume.derval@uclouvain.be

J-C. Régin
University of Nice Sophia-Antipolis, France
E-mail: jcregin@gmail.com

P. Schaus
UCLouvain, Belgium
E-mail: pierre.schaus@uclouvain.be

1 Introduction

The Bin-Packing constraint [21] models the assignment of a set of weighted items to a set of bins. More exactly

$$\text{BinPacking}([X_0, \ldots, X_{n-1}], [w_0, \ldots, w_{n-1}], [L_0, \ldots, L_{m-1}])$$

with X_i the bin to which the item i is assigned, w_i the weight of this item and L_j the load of the bin j. The constraint enforces that $\forall j \in [0, m-1] : L_j = \sum_{i|X_i=j} w_i$.

Domain consistency filtering for Bin-Packing constraints is NP-hard. Hence the community has worked on filtering algorithms based on relaxations following three main directions:

- The problem can be viewed as a combination of one knapsack problems for each bin [21]. However since it does not consider links between the bins, the filtering poorly prunes when there are many items per bin [16].
- A natural way to see the problem is the *transportation model* [18]. The transportation model uses network flows, where each item acts as a source with capacity equal to its weight, and each bin acts as a sink. This model captures the interactions between items and bins, but since items are allowed to be cut, the relaxation is quite poor.
- Another model, the *assignation model*, also uses the flow view of the problem, but does not allow the item to be cut, at the expense of losing information about the weights. To do so, it introduces redundant cardinalities. This view is very similar to the one used by the filtering of the Global Cardinality Constraint [17].

This last model, also known as the *cardinality reasoning* method, is interesting on problems dominated by the assignment aspects of bin-packing. It has shown interesting results in [20].

In the following sections, we describe a method to use the item weight information directly in the *assignation model*, by using the concept of minimum-cost flows, to compute bounds on loads and cardinalities of the bins. We then introduce a new kind of reasoning, called "*too-big/too-small*", which permits to remove candidate items from bins.

2 Definitions and related works

Multiple propagators exist for the Bin-Packing constraint, notably the one from Shaw [21] which attempts to filter domains of the X_i variables using a knapsack formulation. Some work attempted to develop inconsistency check relying on standard bin-packing lower bounds [3]. Finally, a linear programming Arc-flow reformulation was proposed in [2]. Multiple lower and upper bounds have been found for the Bin-Packing problem, such as [9, 10].

Bin-Packing with Cardinality (BPC) was introduced in [20, 15] as an extension to the Bin-Packing constraint:

$$\text{BinPacking}([X_0, \dots, X_{n-1}], [w_0, \dots, w_{n-1}], [L_0, \dots, L_{m-1}], [C_0, \dots, C_{m-1}])$$

with variable C_j the cardinality (number of items assigned) of bin j. It additionally enforces that $\forall j \in [0, m-1] : C_j = |\{i \mid X_i = j\}|$.

The BPC constraint is modeled in [20] decomposing it as a standard BinPacking and a Global Cardinality Constraint (GCC) [17]. An additional simple algorithm computes a lower bound on the loads using the cardinalities, which we recall in section 2.1 improving the communication between the GCC and BinPacking. As shown experimentally in [20], even for bin-packing problems that are initially not constrained by cardinalities this combination can bring additional pruning of the search tree.

Pelsser et al. [15] introduce an improved algorithm to further tighten the bounds on the loads and cardinalities. The propagator is recalled briefly in section 2.2.

Other lower and upper bounds for the BPC (or similar problems that can represent BPC) are presented in [7–9, 11].

In the following sections, we consider that, without loss of generality, the items are ordered by decreasing weight: $w_i \geq w_j$ if $i < j$. We also denote by $\overline{Y}$ and $\underline{Y}$ respectively the upper and lower bound of a given integer variable Y. $\text{dom}(Y)$ is the set representing the domain of Y, i.e. the values to which Y can be assigned.

Definition 1 Let packed_j be the set of items already packed in the bin j, and let cand_j be the set of items that can be assigned to the bin j:

$$\mathit{packed}_j = \{i \mid \text{dom}(X_i) = \{j\}\} \qquad \mathit{cand}_j = \{i \mid j \in \text{dom}(X_i) \wedge |\text{dom}(X_i)| > 1\}$$

We also define $\text{sum}(S) = \sum_{i \in S} w_i$ the sum of weights of items in set S.

2.1 Lower bound on the loads: the SimpleBPC propagator

A lower bound on the cardinality is introduced by Schaus et al.[20]. For each bin, it finds a minimum cardinality subset A_j of cand_j such that $\text{sum}(A_j) \geq \underline{L_j} - \text{sum}(\mathit{packed}_j)$. The update rule on the cardinality lower bound is:

$$\underline{C_j} \leftarrow \max(\underline{C_j}, |\mathit{packed}_j| + |A_j|)$$

Similarly, after computing the maximum cardinality subset B_j such that $\text{sum}(B_j) \leq \overline{L_j} - \text{sum}(\mathit{packed}_j)$, we have that

$$\overline{C_j} \leftarrow \min(\overline{C_j}, |\mathit{packed}_j| + |B_j|)$$

A_j and B_j are computed greedily. For example, for A_j, with the items taken sorted by decreasing weight, the algorithm simply computes the running sum of weights until it overflows $\underline{L_j}$. The SimpleBPC propagator runs in $\mathcal{O}(nm)$, as it needs to visit cand_j for each bin j.

2.2 Pelsser's propagator

The contribution of Pelsser et al.[15] is twofold: they introduce a new upper and lower bound for L_j and provide a sharper way to compute A_j and B_j, using the information of the possible attributions of items to other bins. The new bounds are:

$$\underline{L_j} \leftarrow \max(\underline{L_j}, \mathrm{sum}(packed_j) + \mathrm{sum}(E_j))$$

$$\overline{L_j} \leftarrow \min(\overline{L_j}, \mathrm{sum}(packed_j) + \mathrm{sum}(F_j))$$

with E_j (resp. F_j) the $\underline{C_j} - |packed_j|$ (resp. $\overline{C_j} - |packed_j|$) first items of minimum (resp. maximum) weight assignable together to bin j.

A_j, B_j, E_j and F_j are not computed using the previously presented greedy algorithm. The propagator takes into account the other bins, by using only items that are not required by other bins to fulfill their own capacity requirements.

More precisely, while updating the bounds on a bin j, the algorithm maintains an array *availableForBin*, where initial value for $availableForBin_k$ is $|cand_k| - (\underline{C_k} - |packed_k|)$, which can be viewed as the number of items that the bin k "doesn't need" to fulfill its cardinality lower bound $\underline{C_k}$. Items are then selected greedily, in the same way as the previous algorithm, but an item i can only be taken if

$$\forall k \in \mathrm{dom}(X_i) \wedge k \neq j : availableForBin_k > 0$$

that is, no other bin k *needs* object i to fulfill its own lower cardinality requirement $\underline{C_k}$. In case the condition is not met, this item is not used and the next (larger) one is considered. Each time an item i is taken (for A_j, B_j, E_j or F_j), *availableForBin* is updated accordingly:

$$availableForBin_k \leftarrow availableForBin_k - \begin{cases} 1 \text{ if } k \in \mathrm{dom}(X_i) \\ 0 \text{ otherwise} \end{cases}$$

Example 1 Let us compute the set F_a for the BPC instance presented in Figure 1. F_a should contain the four heaviest items assignable to bin a, without violating bounds requirements of other bins. The initial values in *availableForBin* are $(b = 2, c = 3, d = 2)$.

Items are visited in decreasing weight order. The item of weight 6 can be added to F_a; *availableForBin* is updated accordingly, its new value being $(b = 1, c = 2, d = 2)$. Then items[1] 5 and 4 can also be taken, leading to values $(b = 0, c = 0, d = 2)$.

Therefore the item of weight 3 cannot be taken as $availableForBin_b = 0$. Assigning it in addition to the previously selected item would break the cardinality requirement of bin b. Similarly, item 2 cannot be taken as $availableForBin_c = 0$.

[1] For simplicity, we use in the remaining of this paper items weights as a way to identify items

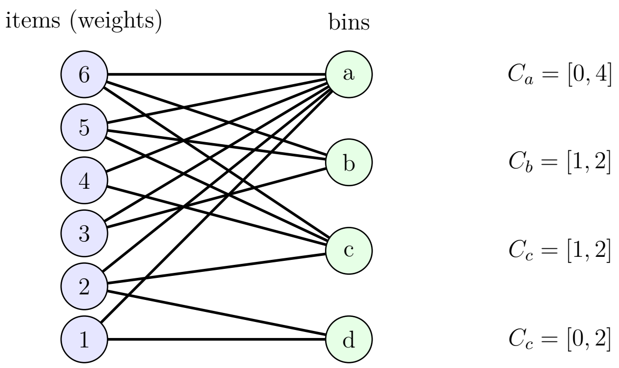


Fig. 1: BPC instance for Example 1. Weights associated with items are shown in their respective nodes.

Item 1 fulfills the requirements to be included in F_a, as *availableForBin*$_d$ = 1. We obtain $F_a = \{6, 5, 4, 1\}$, which provides a better bound ($\overline{L_a} \leq 16$) than the one SimpleBPC obtains ($\overline{L_a} \leq 18$).

Additionally, if, when filling A_j (resp. E_j), the minimum load requirement $\underline{L_j}$ (resp. cardinality requirement $\underline{C_j}$) is not reached, the problem is unfeasible.

Compared to SimpleBPC, verifying *availableForBin* each time an item is added makes Pelsser's propagator run in $\mathcal{O}(nm^2)$.

3 Using flow reasoning on the BPC

In this section, we present a new propagator, called BPCFlow, that subsumes the one from Pelsser et al.[2], by using the same lower and upper bounds for L and C, but uses a flow reasoning inherited from propagators of the Global Cardinality Constraint (GCC)[17] to compute the various sets A_j, B_j, E_j and F_j. Note that GCC is in fact a particular case of Bin-Packing, where all items' weights are unitary.

We first introduce a decomposition of the BinPackingCardinality constraint, using a new constraint called IsolatedBPC, which is the one that BPCFlow solves. We then recall some flow theory background. Section 3.3 introduces a representation of the IsolatedBPC constraint in the form of a network flow, by extending the representation used in Global Cardinality Constraint filtering [17]. Then, the algorithms part of BPCFlow are introduced.

[2] which itself subsumes SimpleBPC

3.1 IsolatedBinPackingCardinality constraint

In order to pose a theoretical foundation on the flow representation of the problem, we introduce a new constraint, IsolatedBPC:

$$\begin{aligned}
&\text{IsolatedBPC}([X_0,\dots,X_{n-1}],[w_0,\dots,w_{n-1}],j,L_j,[C_0,\dots,C_{m-1}]) \equiv \\
&\qquad\qquad L_j = \textstyle\sum_{i|X_i=j} w_i \qquad \text{(Load in isolation)} \\
&\qquad\qquad \forall k \in [0,m-1] : C_k = |\{i \mid X_i = k\}| \quad \text{(GCC constraint)}
\end{aligned}$$

Bin-Packing with Cardinality is decomposed with IsolatedBPC constraints:

$$\begin{aligned}
&\text{BinPacking}([X_0,\dots,X_{n-1}],[w_0,\dots,w_{n-1}],[L_0,\dots,L_{m-1}],[C_0,\dots,C_{m-1}]) \equiv \\
&\bigwedge_{j\in[0,m-1]} \text{IsolatedBPC}([X_0,\dots,X_{n-1}],[w_0,\dots,w_{n-1}],j,L_j,[C_0,\dots,C_{m-1}])
\end{aligned}$$

3.2 Flow theory background

We recall the basic concepts of network-flow theory [4–6].

Definition 2 A **flow network** G is an oriented graph, for which each edge (u,v) is additionally associated with a lower and upper capacity, called respectively $\text{low}(u,v)$ and $\text{up}(u,v)$. A **flow** $f(u,v)$ is a function which represents the *flow* going from a vertex u to another vertex v. If the edge (u,v) does not exist, then $\text{low}(u,v) = \text{up}(u,v) = f(u,v) = 0$. A flow must additionally respects the conservation law:

$$\forall u : \sum_v f(u,v) = \sum_v f(v,u)$$

That is, the amount of flow *entering* into a node u must be equal to the amount of flow *exiting* it.

Definition 3 A **valid** flow f is a flow which respects the lower and upper capacity of each edge: $\forall u,v : \text{low}(u,v) \leq f(u,v) \leq \text{up}(u,v)$. A **maximum** flow for an edge (u,v) is a flow f that maximizes the flow value on the edge linking nodes u and v.

For simplicity of representation, we use in the remaining of this paper two special nodes: the source s and the sink t, which are linked by an edge with $\text{low}(t,s) = 0$ and $\text{up}(t,s) = +\infty$. This edge is most of the time implied, and when no indication on which edge the flow is maximal, it is always between t and s.

Definition 4 A **weighted flow network** is a regular flow network that associates with each edge (u,v) a cost **per unit of flow** denoted $p(u,v)$. The total cost of a flow is then:

$$\text{cost}(f) = \sum_{u,v} f(u,v) \cdot p(u,v)$$

A **minimum cost maximum flow** f is a flow that is maximum, and that has the minimum possible cost (i.e. there does not exist another maximum flow f' such that $\text{cost}(f) > \text{cost}(f')$). Max-cost max-flow and min-cost min-flow are defined in a similar way.

Definition 5 The **residual graph** of a flow network G and of a flow f is noted $R_G(f)$. It is composed of the same vertices as the original graph, and its edges are defined as follows: $\forall$ edge $(u, v) \in G$,

- If $f(u, v) < \text{up}(u, v)$, then $(u, v) \in R_G(f)$, and two values are associated with this edge: its capacity $\text{up}'(u, v) = \text{up}(u, v) - f(u, v)$ and its cost $p'(u, v) = p(u, v)$
- If $f(u, v) > \text{low}(u, v)$, then $(v, u) \in R_G(f)$, and two values are associated with this edge: its capacity $\text{up}'(v, u) = f(u, v) - \text{low}(u, v)$ and its cost $p'(v, u) = -p(u, v)$

An **augmenting path** of capacity a in a residual graph $R_G(f)$ is a path such that the minimum capacity of any edge in the path is a. The weight of path/cycle in $R_G(f)$ is the sum of the cost of the edges in this path/cycle. A **negative (positive) cycle** is a cycle with negative (positive) weight.

These concepts of residual graphs and augmenting paths, central in flow theory, lead to very important results, which are the basis of the following section.

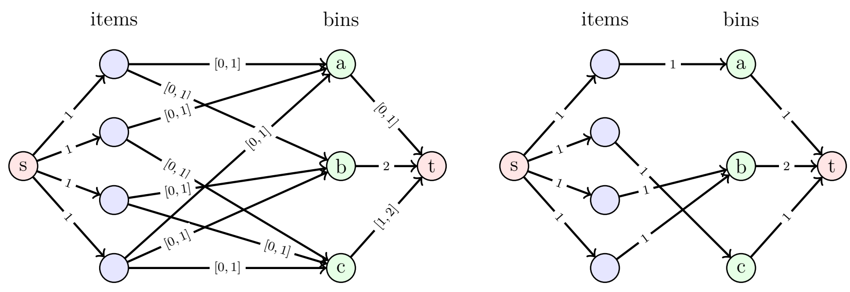
Theorem 1 *[5] A flow f is maximal between two nodes t and s if and only if there is no augmenting path in $R_G(f)$ from s to t.*

Theorem 2 *[6] A flow between two nodes t and s is of minimum cost if and only if there is no (strictly) negative cycle in $R_G(f)$. It is of maximum cost if and only if there is no (strictly) positive cycle in $R_G(f)$.*

These two theorems allow to compute maximum flows and min-cost flows easily, by iteratively finding augmenting paths from s to t (to find a maximum flow) or by iteratively finding negatively weighted cycles, and *canceling* them (i.e. maximizing the flow along the cycle to reduce the overall cost). See [5] and [6] for more details about the algorithms used.

3.3 An IsolatedBPC representation as a flow network

Régin [17] shows that a GCC can be represented as a flow network with a one-to-one correspondence between feasible flows and feasible solutions. A bipartite graph of *values* (that we call *bins* in a bin-packing problem) and *variables* (that we call *items*), with an edge linking item i and bin j iff $j \in \text{dom}(X_i)$ is first created. Then a source s and a sink t are added, such that the source is linked to each item i with an edge $\text{low}(s, i) = \text{up}(s, i) = 1$, and each bin j is linked to the sink with an edge of capacity bounds $\text{low}(j, t) = \underline{C_j}$ and $\text{up}(j, t) = \overline{C_j}$. All edges from items to bins have $\text{low}(i, j) = 0$ and $\text{up}(i, j) = 1$



(a) Flow network (b) An example of valid flow

Fig. 2: Example 2

Example 2 Given a bin-packing problem, with four items that can be assigned respectively to bins $\{a, b\}$, $\{a, c\}$, $\{b, c\}$ and $\{a, b, c\}$, such that: $\underline{C_a} = 0$, $\overline{C_a} = 1$, $\underline{C_b} = 2$, $\overline{C_b} = 2$, $\underline{C_c} = 1$, $\overline{C_c} = 2$. We obtain the network flow represented in Figure 2a. A valid flow is represented in Figure 2b.

Régin then uses this representation to reach Generalized Arc Consistency[12]. Typical implementations of the propagators extend the results of Régin by adding a shaving on the bin cardinality bounds, by computing a minimum and maximum flow on each edge from bins to the sink, reaching bound(Z) consistency [1] on C.

Our first contribution, presented in the remaining of this section, is an extension of this representation (called the *assignation model*) for the bin-packing problem, by transforming it into a max-flow max-cost problem, and associated methods to infer bounds for cardinalities and loads.

We extend the assignation model for the bin-packing by creating, for each bin j, a weighted flow network G_j such that:

- the edges from G (the previously described graph for the equivalent GCC) are conserved, including their lower and upper capacity;
- each edge between an item i and the targeted bin j has a cost $p(i, j) = w_i$;
- all the other edges have a cost of zero.

By construction, at most one edge for each item has a non-zero cost, and all non-zero cost edges are linked to bin j.

Example 3 Reusing the Example 2, such that items have respectively weights 10, 5, 12 and 2, the Figure 3 represents the graph G_b. The total cost of the flow represented in Figure 2b is 14.

Similarly as for GCC, every solution to the IsolatedBPC constraint for bin j can be converted into a valid flow of cost L_j in G_j and vice-versa.

In the following subsections, we introduce a new propagator based on this representation that we call BPCFlow. For the sake of readability, any flow that appears in the following section is considered *valid*. Such flows have $f(t, s) = n$ (every item is assigned to exactly one bin) by construction of G_j.

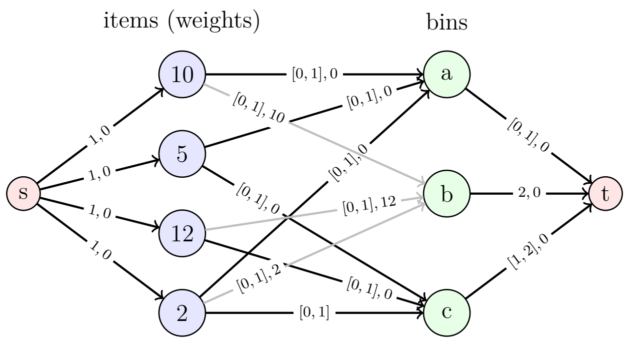


Fig. 3: Weighted flow network for example 3 and bin 1. The notation "$[low, up], p$" on an edge indicates both its capacity bounds and its cost p. Non-zero cost edges are in gray. Items weights are represented into the items respective nodes.

3.4 Computing bounds for the loads

This representation G_i allows us to bound L_j, by computing a maximum-cost flow $f_{j,M}$ and a minimum-cost flow $f_{j,m}$ (between t and s). The filtering rule is thus:

$$\underline{L_j} \leftarrow \max(\underline{L_j}, \text{cost}(f_{j,m})) \qquad \overline{L_j} \leftarrow \min(\overline{L_j}, \text{cost}(f_{j,M}))$$

This is similar to the bounds in Pessler's propagator but with the definition of E_j and F_j based on the edges used in the minimum and maximum flows:

$$E_j = \{i \mid f_{j,m}(i,j) = 1 \land i \notin packed_j\} \qquad F_j = \{i \mid f_{j,M}(i,j) = 1 \land i \notin packed_j\}$$

For convenience, we compute the max-cost max-flow between bin j and the sink rather than between the sink and the source. Let us proof that this is indeed equivalent:

Theorem 3 *In G_j, any maximum-cost maximum-flow $f_{j,MM}$ between j and t is also a maximum-cost flow from t to s.*

Proof Since $f_{j,MM}$ is valid, is it maximal from t to s with $f(t,s) = n$. As the only edges that have a non-zero cost are the edges linked to bin j, optimizing the cost between t and s, or between bin j and t is equivalent. Finally, we note that $f_{j,MM}$ is not only a maximum flow with the maximum cost between j and t, but also a maximum-cost flow among all the valid flows in G_j, as, by construction of G_j, there is no positive cycle allowing to increase the cost by diminishing the flow. □

Algorithm 1 describes how to compute a maximum-cost flow in G_j starting from any valid flow. For a given bin j, each item is considered in decreasing weight order, and tentatively assigned to bin j, with the restriction that unassigning heavier items is forbidden. In case it is not possible, this item is not assigned to bin j. In Algorithm 1, when item i is assigned to bin j, the flow in this edge cannot be canceled in next iterations. This is achieved by removing those edges from the residual graph $(R_{G_j}(f) - S)$ when looking for the next simple path.

Algorithm 1 `ComputeMaxCostFlow` (helper function for BPCFlow)

Require: f, a valid flow for G_j,
  $X_0, ..., X_{n-1}, C_0, ..., C_{m-1}, L_j, j$ the arguments of the IsolatedBPC constraint.
Ensure: f is a maximum-cost flow at the end of the algorithm call
  $S \leftarrow \emptyset$
  for all item i s.t. $j \in X_i$ in decreasing order of weight **do**
    if i is not assigned to j and $\exists$ a simple path p from bin j to item i in $R_{G_j}(f) - S$ **then**
      for all edge (item u, bin v) in p **do**
        Unassign item u from its previously assigned bin
        Assign item u to bin v
      end for
    end if
    $S \leftarrow S \cup \{(j, i)\}$
  end for
  return f

The justification of why the Algorithm 1 works is given in the following theorem:

Theorem 4 *Given a valid flow f in network G_j, let the cycle p be the positively weighted cycle in $R_{G_j}(f)$ that contains the heaviest possible item i_0 (among all the available cycles) such that the edge (i_0, j) is part of the cycle.*[3] *Then, after cancellation, no positively weighted cycle contains an edge (k, j) such that $w_k > w_{i_0}$.*

Proof Without loss of generality, let us consider only cycles that starts and ends at node j, without visiting it during the cycle. Let us consider, by contradiction, that there exists a positively weighted cycle p_0 in $R_{G_j}(f)$ such that its cancellation assigns item i_0 to bin j and creates a cycle p_1 such that its cancellation assigns item i_1 to bin j, with $w_{i_1} > w_{i_0}$.

Each of these two cycles either unassigns an item, or makes use of the edge (j, t), increasing the overall cardinality of bin j. We name the (single) item unassigned or, in the other case, the node t, as k_0 for p_0 and k_1 for p_1. For simplicity, we pose that $w_t = 0$. Since the cycles are positively weighted, $w_{i_0} > w_{k_0}$ and $w_{i_1} > w_{k_1}$.

By construction, there exists two nodes u and v (which can be either bins or items) such that p_0 is in the form $(j, k_0, \cdots, u, \cdots, v, \cdots, i_0, j)$ and p_1 is in the form $(j, k_1, \cdots, v, \cdots, u, \cdots, i_1, j)$.[4] Then, the cycle $p_2 = (j, k_0, u, \cdots, i_1, j)$ already exists in $R_{G_j}(f)$ and is positively weighted since $w_{i_1} > w_{i_0} > w_{k_0}$.

[3] i. e. item i_0 will be assigned to bin j after cancellation of the cycle

[4] Note that u and v can be confounded with i_0, k_0, i_1 or k_1, which does not change the proof.

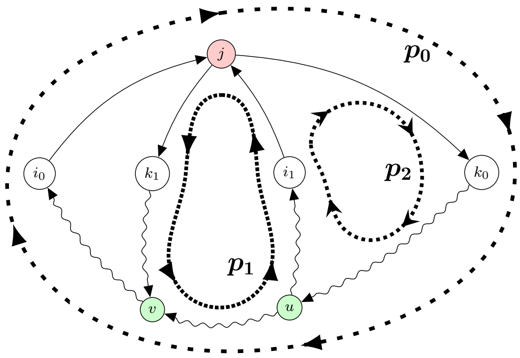


Fig. 4: If inverting (/canceling) the cycle p_0 creates a new cycle p_1, then there was previously an existing cycle p_2 containing the end of p_1 and the beginning of p_0 (from j).

Thus i_0 is not the maximal item available to create a cycle, leading to a contradiction. See Figure 4 for a visual explanation. □

Said differently, once the maximum weighted available item is assigned to the bin by cycle cancellation, no further iteration of the cycle-finding algorithm will unassign this item for a heavier one. Finding a cycle consists in running a Depth-First-Search in $\mathcal{O}(nm)$. [5] Thus, the complexity of `ComputeMaxCostFlow` is $\mathcal{O}(nm^2)$, which is strongly polynomial in the size of the input. We can define `ComputeMinCostFlow` similarly, by searching only negative cycles, with items ordered by increasing weight.

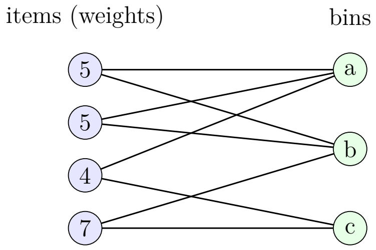
Algorithm 2 describes how to compute the bounds for L_j. As [17] proves, the complexity to find the maximum (/minimum) flow in this representation is $\mathcal{O}(m^3)$. The complexity of Algorithm 2 is then $\mathcal{O}(m^3 + m^2 n)$.

Algorithm 2 Compute bounds for L_j (first part of BPCFlow)

$f \leftarrow$ valid flow in G_j
$\underline{L_j} = \max(\underline{L_j}, \text{cost}(\texttt{ComputeMinCostFlow}(\text{f})))$
$\overline{L_j} = \min(\overline{L_j}, \text{cost}(\texttt{ComputeMaxCostFlow}(\text{f})))$

Example 4 Figure 5 shows an instance where Pelsser's propagator does not improve bounds on bin loads as much as BPCFlow does: for bin a, the minimum-cost flow (which is a lower bound for L_a) is 5, as the item 4 cannot be taken alone.

[5] the worst case being a complete graph



Bin		Initial	Pelsser	BPCFlow
a	L_a	$[0, 10]$	$[4, 10]$	**[5, 10]**
	C_a	$[1, 2]$	-	-
b	L_b	$[0, 7]$	$[4, 7]$	$[4, 7]$
	C_b	$[0, 1]$	-	-
c	L_c	$[0, 11]$	$[4, 11]$	$[4, 11]$
	C_c	$[1, 2]$	-	-

Fig. 5: BPC instance for example 4. The table on the right shows the evolution of loads and cardinality bounds depending on the filtering used. Value ”−” means that the value remained the same as the initial one.

3.5 Computing bounds for the cardinalities

Similarly to Pessler's propagator, an upper bound for C is the value of the maximum-valued minimum-cost flow whose cost is lesser than $\overline{L}$. A lower bound for C can be defined similarly using $\underline{L}$.

Algorithm 3 finds the minimum cardinality for bin j such that the (maximum) cost of the flow (the load) is above the minimum load, $\underline{L_j}$, using a dichotomic search. The algorithm can, of course, be modified to compute $\overline{C_j}$ instead of $\underline{C_j}$, simply by starting from a minimum-cost minimum flow and searching the cardinality for which the minimum cost flow value is directly below the maximum $\overline{L_j}$.

The algorithm modifies the minimum/maximal cardinality of bin j dynamically, via $\text{up}(j, t)$ and $\text{low}(j, t)$, in order to modify the value of the flow. The function `FixFlow` corrects the flow accordingly. If Δ is the absolute difference between the values of $\text{up}(j, t)$ or $\text{low}(j, t)$ compared to their old values, its complexity is $\mathcal{O}(\Delta nm)$ as at most Δ augmenting paths are needed to correct the flow. `FixFlow` returns false when it is impossible to recreate a valid flow. The algorithm uses:

- one initial call to the max-flow algorithm, in $\mathcal{O}(m^3)$;
- $\mathcal{O}(\log(\overline{C_j} - \underline{C_j})) \in \mathcal{O}(\log n)$ calls to `ComputeMaxCostFlow`(j), leading to a complexity of $\mathcal{O}(m^2 n \log(n))$;
- the sum of the modification between calls of `FixFlow`(j) is dominated by $\sum_{i=1} \frac{n}{2^i} \in \mathcal{O}(n)$, thus the calls have a total complexity of $\mathcal{O}(n^2 m)$.

The complexity of Algorithm 3 is then $\mathcal{O}(m^3 + m^2 n \log(n) + n^2 m)$. [6] The combination of both Algorithms 2 and 3 forms BPCFlow, which has the same complexity as Algorithm 3.

Example 5 Figure 6 shows another BPC instance (which conserves the graph from Example 4 but has different load/cardinality bounds), on which the com-

[6] A small variation of this algorithm, where the dichotomic search is replaced by a linear one, leading to a complexity of $\mathcal{O}(m^3 + m^2 n^2)$, was also implemented. Its performances are very close to its dichotomic counterpart.

Algorithm 3 Compute bounds for $\underline{C_j}$ (second part of BPCFlow)

up$(j,t) \leftarrow \overline{C_j}$, low$(j,t) \leftarrow \overline{C_j}$ ▷*Start from a maximum-cost maximum flow*
curLoad $\leftarrow$ `ComputeMaxCostFlow`(j) ▷*The current flow is maximum in j*

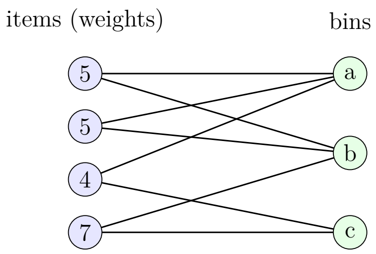
dichoMaxCard $\leftarrow \overline{C_j}$ ▷*Minimum "valid" card. at any point of the dichotomic search*
dichoMaxCardLoad $\leftarrow$ curLoad ▷*Store the best (lower) reached load*
dichoMinCard $\leftarrow \underline{C_j} - 1$ ▷*Not reached*

while dichoMinCard $+ 1 \neq$ dichoMaxCard **do**
  attempt $\leftarrow \frac{\text{dichoMinCard}+\text{dichoMaxCard}}{2}$ ▷*Flow value to be tested*
  up$(j,t) \leftarrow$ attempt, low$(j,t) \leftarrow$ attempt

  if `FixFlow`(j) **then** ▷*Flow is valid, check cost*
    curLoad $\leftarrow$ `ComputeMaxCostFlow`(j)
    if curLoad $\geq \underline{L_j}$ **then** ▷*Load is above minimum*
      dichoMaxCard $\leftarrow$ attempt
      dichoMaxCardLoad $\leftarrow$ curLoad
    else ▷*Load breaks the requirement*
      dichoMinCard $\leftarrow$ attempt
    end if
  else ▷*A valid flow was not found*
    dichoMinCard $\leftarrow$ attempt
  end if
end while

$\underline{C_j} \leftarrow$ dichoMaxCard ▷*Update with the newly computed bound*

up$(j,t) \leftarrow \overline{C_j}$, low$(j,t) \leftarrow \underline{C_j}$ ▷*Restore bounds on the bin*
`FixFlow`(j) ▷*Ensure we have a valid flow at the end*



Bin		Initial	Pelsser	BPCFlow
a	L_a	$[0, 10]$	-	**[5, 10]**
	C_a	$[0, 2]$	-	**[1, 2]**
b	L_b	$[0, 7]$	-	-
	C_b	$[0, 2]$	$[0, 1]$	$[0, 1]$
c	L_c	$[0, 11]$	-	**[4, 11]**
	C_c	$[0, 2]$	-	**[1, 2]**

Fig. 6: BPC instance for example 5

bination of algorithms 2 and 3 to prune both loads and cardinalities allows BPCFlow to prune more than Pelsser's method. It is indeed impossible to create a valid flow with no flow to bin a or c, thus their cardinality is at most one. As example 4, item with weight 4 cannot be taken alone in bin a (there is no such flow), and since one item must be taken in bin c, the lower load bound for this bin is 4 (lightest item that can be taken).

4 Filtering candidates using load and cardinality information

The filtering introduced in [15,20] only tightens the bounds of the cardinality and load variables relying on the GCC to filter the domain of item variables X_i. This section introduces a filtering on the domains of variables X_i using a too-big/too-small reasoning: an item i cannot be assigned to a bin j if it is too big (resp. too small), i.e. there does not exist a set of items including i such that the load is lighter than $\overline{L_j}$ (resp. heavier than $\underline{L_j}$). We propose three variations of this method, based on the techniques presented in the previous sections. We denote by lightest(k, S) (resp. heaviest(k, S)) the subset of S composed of the k lightest (resp. heaviest) items in S.

- **SimpleBPC+**. For item i we can determine if it is too big by selecting the $\underline{C_j} - 1$ lightest items (different from i). If the weight of the item plus the selected items is greater than the upper load bound $\overline{L_j}$, the item cannot be assigned to bin j.
  The detection of too small items is done similarly, by finding the $\overline{C_j} - 1$ heaviest items and verifying that the sum of weights is not below $\underline{L_j}$. Taking $packed_j$ into account the *too-big/too-small* filtering rules are:

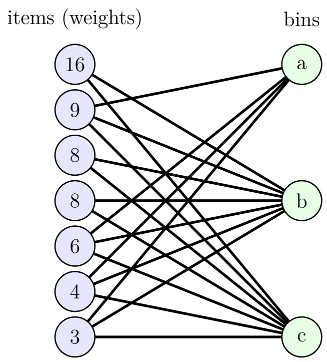
$$w_i + \text{sum}(\text{lightest}(\underline{C_j} - |packed_j| - 1, cand_j \setminus \{i\})) > \overline{L_j} - \text{sum}(packed_j) \implies X_i \neq j \quad \text{(too big)}$$

$$w_i + \text{sum}(\text{heaviest}(\overline{C_j} - |packed_j| - 1, cand_j \setminus \{i\})) < \underline{L_j} - \text{sum}(packed_j) \implies X_i \neq j \quad \text{(too small)}$$

  Example 6 Figure 7a presents a BPC instance which is consistent with respect to BPCFlow. Bin a must have a cardinality of two; applying the "too big" rule above, we find the lightest set of cardinality $C_a - 1 = 1$, which is the set containing only the item 3. We then conclude that the item 9 cannot be assigned to bin a, as the minimum cardinality requirement imposes to take two items, and that item 9 cannot fit with 3, the lightest item $(3 + 9 > \underline{L_a})$. Propagating this modification to the other constraints leads to the assignation of item 6 to bin a.
  Figure 7c shows the instance, now consistent with both SimpleBPC+ and BPCFlow.

- **Pelsser+**. Similarly, we can determine the set of the $\underline{C_j} - 1$ lightest items using the Pelsser's method from section 2.2 (and similarly for the $\overline{C_j} - 1$ heaviest items).

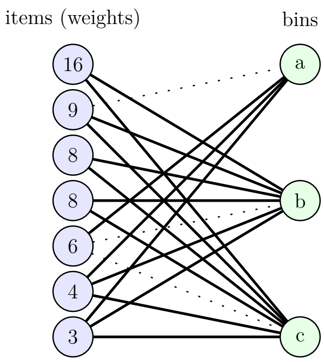
  Example 7 From instance presented in Figure 7a, Pelsser+ firstly prunes item 9 from bin a, similarly to SimpleBPC+, and the propagation assigns item 6 to bin a.
  From there, we can use the "too big" rule on bin c. Let us compute the minimum weighted assignable set of size $C_c - 1 = 2$ using the Pelsser's rule. Item 3 can be taken, but item 4 cannot, as it would forbid bin a to fullfill



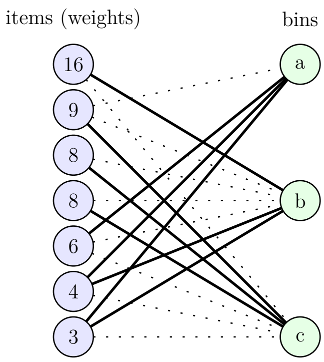
(a) An instance of BPC, consistent with BPCFlow

Bin		Initial (BPCFlow)	SimpleBPC+	Pelsser+
a	L_a	$[9, 10]$	-	-
	C_a	2	-	-
b	L_b	$[18, 20]$	-	$[19 - 20]$
	C_b	2	-	-
c	L_c	$[24, 26]$	-	25
	C_c	3	-	-

(b) Load and cardinality bounds for various filtering of the original instance



(c) BPC instance presented in Figure 7a, further pruned with SimpleBPC+. Removed edges are represented as dotted lines.



(d) BPC instance presented in figure 7a, further pruned with Pelsser+. Removed edges are represented as dotted lines.

Fig. 7: BPC instance for example 6 and 7

its cardinality requirement (item 3 has been taken, item 4 and 6 are then needed for bin a). Item 8 is the next assignable item. The weight of the minimum weighted set is then 11. According to the "too big" rule, items with weight greater than $\overline{L_c} - 11 = 15$ cannot be assigned to bin c, which leads to the exclusion of item 16.
Now, with the "too small" rule, again for bin c, we compute a maximum weighted set of weight 17 (with items 9 and 8), excluding all items with weight lesser than $\underline{L_c} - 17 = 7$, namely items $3, 4$ and 6. Items 9, 8 and 8 must then be assigned to bin c to fulfill its cardinality requirements.
Figure 7d shows the instance, now consistent with both Pelsser+ and BPCFlow.

– **BPCFlow+**. The flow network presented in the previous section can also be extended to verify if the min-weighted set of cardinality $\underline{C_j}$ and containing a specific item i has a lesser weight than $\overline{L_j}$, and conversely for the max-weighted case.

This last point requires a specific algorithm on the network flow. Given a flow f that is of min-cost on G_j, we can check if an item can be taken (i.e. the min-cost flow containing the item is of cost lesser than $\overline{L_j}$) iff:

1. the item (the edge (i, j)) is already used in flow f;
2. there exists a cycle in $R_{G_j}(f)$ which contains edge (i, j) and whose cost is lesser than $\overline{L_j} - \text{cost}(f)$, i.e. canceling this cycle will not produce a flow with cost greater than $\overline{L_j}$.

A simple way to check this last property is to start a DFS from node j, and to find a path to node i that uses only edges whose costs are lesser than $\overline{L_j} - \text{cost}(f) - w_i$.

Example 8 Figure 8 shows the residual graph of a minimum-cost minimum-flow of a sample BPC problem, for bin j. Let us define that this bin has $\overline{L_j} = 6$. In this context, the items 1 and 2 are not *too big* as they are already in the min-cost flow. Item 3 can be taken, because there exists a cycle (namely $3 \to j \to 2 \to b \to 3$) whose cost is 1, and canceling it would create a new flow of cost $3 \leq \overline{L_j}$. Item 4 cannot be taken into bin j has there is no cycle that includes edge $(4, j)$. There exists a cycle for item 8, but it cannot be used as the cycle total weight is 6, leading to a cost of $9 > \overline{L_j}$.

The too-big/too-small reasoning brings additional filtering to the existing propagators without impacting the asymptotic complexity:

– for SimpleBPC+ and Pelsser+, it only adds an $\mathcal{O}(nm)$ operation (for checking each item for each bin) on the already $\mathcal{O}(nm)$ pruning of SimpleBPC and $\mathcal{O}(nm^2)$ of Pelsser's propagator;
– for BPCFlow+, the additional DFS and checks needed also add $\mathcal{O}(nm)$ to the already $\mathcal{O}(m^3 + m^2 n \log(n) + n^2 m)$ pruning of BPCFlow.

Table 1 is a summary of the propagators presented throughout this paper. Figure 9 shows the subsuming relations between them.

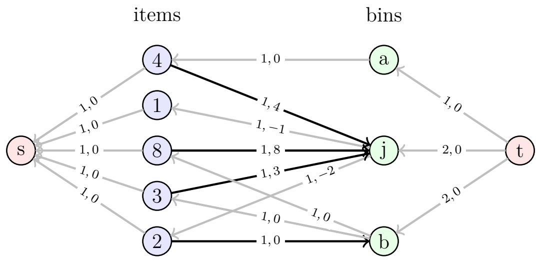


Fig. 8: Residual graph R_{G_j} for Examples 8

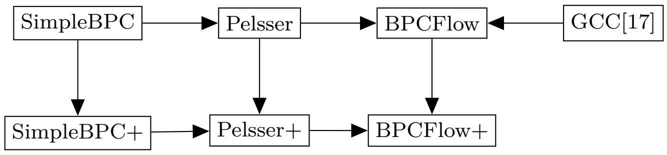


Fig. 9: Relations between propagator implementations. $a \rightarrow b$ means that b subsumes a (the relation obviously holds transitively).

5 Experiments

We focus the experiments on two problems: the Balanced Academic Curriculum Problem (BACP)[13] and the Tank Allocation Problem (TAP) [19]. Schaus et al.[19] only provided a single instance of TAP; a new set of 2592 instances has been generated for this research, with various parameters. While the BACP involves a direct bin-packing with cardinality constraint, the one used in TAP is redundant.

All experiments use the search tree-replay mechanism proposed in [22], which allows to compare propagators strengths without most of the search heuristic influence.

When SimpleBPC(+) and Pelsser(+) are used, an additional GCC constraint is also added, in order to be able to compare results with BPCFlow(+) that includes a similar GCC propagator.

The propagators have been implemented using the OscaR-CP solver, and the source codes are available on the repository of OscaR [14] .

5.1 BPCFlow versus Pelsser's propagator and SimpleBPC

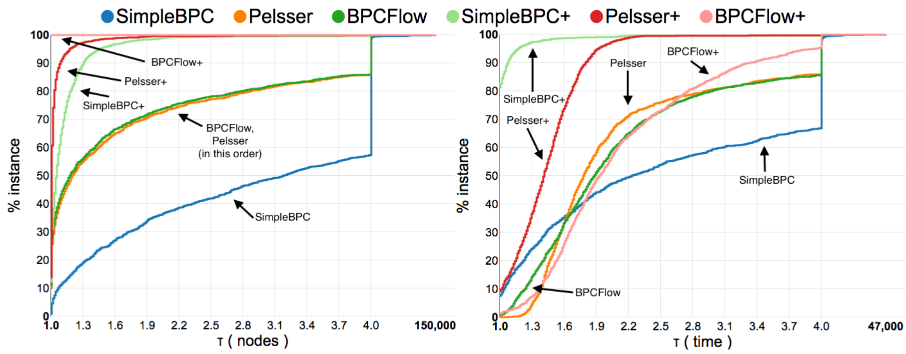
Figure 10 shows various performance profiles for instances of BACP and TAP. A point (x, y) indicates the percentage of instances for which an instance is solved within a time limit of at most y times the best approach on this instance. As shown, on most problems, BPCFlow does not prune significantly more than Pelsser, while consuming roughly the same amount of computation time. Without considering the too-big/too-small methods, using Pelsser

Table 1: Summary of the presented propagators

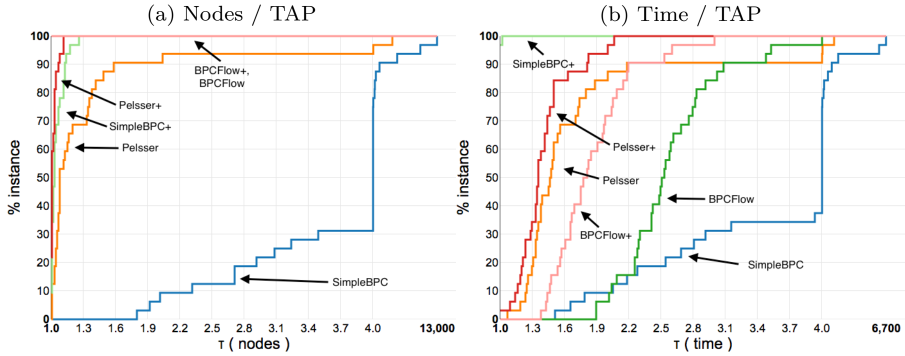
Name	Usage of other bins	Filters loads / cards	Filters item assignations (too-big / too-small reasoning)	Filtering power[a]	Complexity
SimpleBPC [20]	None	yes[b]	no	weak	$\mathcal{O}(nm)$
SimpleBPC+		yes	yes	strong	
Pelsser [15]	Use loads/cards information from other bins; avoid using items when they are needed to fullfill bounds of another bin. Consider items in isolation.	yes	no	moderate	$\mathcal{O}(nm^2)$
Pelsser+			yes	stronger	
BPCFlow	Use loads/cards information from other bins; use min/max cardinality flows to check feasability of bounds. Infer from the whole assignation graph rather than from items in isolation.	yes	no	strong	$\mathcal{O}(m^3 + m^2 n \log(n) + n^2 m)$
BPCFlow+			yes	strongest	

[a] Estimation based on experiments. Recall that, strictly speaking, filtering power of BPCFlow and SimpleBPC+ cannot be compared, altough SimpleBPC+ tend to prune more in our experiments. Same remark goes for the couples SimpleBPC+/Pelsser, Pelsser+/BPCFlow.
[b] Cardinalities are not filtered in the original paper, but the method from Pelsser et al. can be adapted to use the same technique as the original SimpleBPC paper. This is made in some implementations.

is then, most of the time, the best choice. However, this is not true for all instances: BPCFlow can give appreciable speedups compared to Pelsser's propagator on some instances, particularly on instances where relations between items and bins are complex. See Table 2 for a selection of TAP instances where this behavior occurs.



(a) Nodes / TAP (b) Time / TAP



(c) Nodes / BACP (d) Time / BACP

Fig. 10: Performance profiles of presented propagators. Represents the percentage of instances solved depending on the ratio of time taken (in seconds) or of nodes visited, versus the best method. See [22] for details. Instances that never take more than 10 seconds of computation time are not represented.

Table 2: Three best results in number of node visited and in computation time for Pelsser vs BPCFlow, for the instances of TAP that visited more than 100k nodes.

TAP inst. n°	Nodes visited			Computation time (ms)		
	Pelsser	BPCFlow	Gain	Pelsser	BPCFlow	Gain
1757	2418**k**	769**k**	**68.2%**	554.5	309.4	**44.2%**
919	372**k**	139**k**	**62.7%**	35.6	20.4	**42.7%**
895	1651**k**	937**k**	**43.3%**	122.9	70.8	**42.4%**
917	10417**k**	6483**k**	**37.8%**	865.1	513.0	**40.7%**
1558	423**k**	299**k**	**29.2%**	96.4	59.6	**38.2%**
1757	2418**k**	769**k**	**68.2%**	554.5	309.4	**44.2%**
919	372**k**	139**k**	**62.7%**	35.6	20.4	**42.7%**
2305	1481**k**	1455**k**	**1.7%**	546.4	313.1	**42.7%**
895	1651**k**	937**k**	**43.3%**	122.9	70.8	**42.4%**
917	10417**k**	6483**k**	**37.8%**	865.1	513.0	**40.7%**

5.2 Too-big/too-small reasoning

Figure 10 shows that the addition of the too-big/too-small reasoning outperforms the previous propagators, with SimpleBPC+ giving the best speedups, despite being very simple. SimpleBPC+ is the best choice for 91% of BAPC instances, and for 78% of TAP instances. Overall, using too-big/too-small reasoning provides speedups for 98% of BAPC instances, and 87% for TAP ones.

The differences in the amount of visited nodes between SimpleBPC+, Pelsser+ and BPCFlow+ are small but not nonexistent. This shows that even if SimpleBPC+ is the best choice for most BPC instances, other methods are still useful, particularly on very difficult instances, where they drastically reduce computation time. Table 3 shows selected results for each propagator, showing that Pelsser+ and BPCFlow+ still can bring significant gains on some instances.

6 Conclusion

We introduced four new propagators for the Bin-Packing (with cardinality) problem, namely BPCFlow, SimpleBPC+, Pelsser+ and BPCFlow+, based on the assignation model. BPCFlow is based on the usage of weighted flow, and we have shown that previous work, SimpleBPC [20] and Pelsser's propagator [15] are relaxations of this model. Experiments on BPCFlow show that while it allows to improve solving time on very difficult instances, Pelsser is still the best approach for most problems (when not considering the too-big/too-small methods), as it provides a better pruning/computation time compromise.

SimpleBPC+, Pelsser+, and BPCFlow+ are variation based on the too-big/too-small reasoning, which attempts to remove items that are either too heavy, or too light, to be assigned to specific bins. We have shown that this approach outperforms previous works in nearly 88% of the tested instances, notably with the SimpleBPC+ propagator.

Acknowledgements We thank the anonymous reviewer for suggesting the idea of using a dichotomic search in Algorithm 3.

Table 3: Selected results, showing that none of the proposed propagators supersedes all the others. Time is in seconds, means over three runs.

TAP	Node	Time	**Node**	**(gain)**	**Time**	**(gain)**
inst n°	SimpleBPC		**SimpleBPC+**			
381	11013**k**	1172.8	29	(100.0%)	0.006	(100.0%)
2332	5864**k**	1097.6	10	(100.0%)	0.02	(100.0%)
1675	4118**k**	1100.7	56	(100.0%)	0.036	(100.0%)
1563	172**k**	32.5	172**k**	(0.19%)	40.0	(-23.4%)
	Pelsser		**Pelsser+**			
2324	724**k**	240.8	17	(100.0%)	0.026	(100.0%)
2003	953**k**	98.6	61	(100.0%)	0.02	(100.0%)
1062	4552**k**	768.3	3540	(99.9%)	0.246	(100.0%)
2402	7577**k**	2142.9	7171**k**	(5.4%)	2822.0	(-31.7%)
	BPCFlow		**BPCFlow+**			
1727	8768**k**	3314.3	0	(100.0%)	0.0	(100.0%)
2324	724**k**	324.9	17	(100.0%)	0.043	(100.0%)
2003	934**k**	95.3	61	(100.0%)	0.036	(100.0%)
1905	6285**k**	1063.8	6285**k**	(0.0%)	2982.9	(-180.4%)
	SimpleBPC+		**Pelsser+**			
2197	137**k**	15.8	44**k**	(68.2%)	6.3	(60.1%)
919	275**k**	22.1	114**k**	(58.5%)	10.0	(54.9%)
918	260**k**	32.3	161**k**	(37.9%)	15.3	(52.6%)
2402	7234**k**	1315.7	7171**k**	(0.87%)	2822.0	(-114.5%)
	SimpleBPC+		**BPCFlow+**			
2033	6801**k**	1240.5	0	(100.0%)	0.0	(100.0%)
2197	137**k**	15.8	9927	(92.7%)	0.793	(95.0%)
1756	58**k**	12.6	3436	(94.1%)	0.883	(93.0%)
2194	7844**k**	941.7	7844**k**	(0.0%)	4914.2	(-421.8%)
	Pelsser+		**BPCFlow+**			
2042	9632**k**	2439.1	0	(100.0%)	0.0	(100.0%)
1756	57**k**	13.6	3436	(94.0%)	0.883	(93.5%)
2081	75**k**	10.2	19**k**	(74.9%)	1.2	(88.2%)
2194	7844**k**	1526.0	7844**k**	(0.0%)	4914.2	(-222.0%)

References

1. Bessiere, C.: Constraint propagation. Foundations of Artificial Intelligence **2**, 29–83 (2006)
2. Cambazard, H., O'Sullivan, B.: Propagating the bin packing constraint using linear programming. In: D. Cohen (ed.) International Conference on Principles and Practice of Constraint Programming, pp. 129–136. Springer, Berlin, Heidelberg (2010)
3. Dupuis, J., Schaus, P., Deville, Y.: Consistency check for the bin packing constraint revisited. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming, pp. 117–122. Springer (2010)

4. Edmonds, J., Karp, R. M.: Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM (JACM) **19**(2), 248–264 (1972)
5. Ford Jr, L. R., Fulkerson, D. R.: A simple algorithm for finding maximal network flows and an application to the Hitchcock problem. Tech. rep., DTIC Document (1955)
6. Goldberg, A. V., Tarjan, R. E.: Finding minimum-cost circulations by canceling negative cycles. J. ACM **36**(4), 873–886 (1989)
7. Labbé, M., Laporte, G., Martello, S.: An exact algorithm for the dual bin packing problem. Operations Research Letters **17**(1), 9 – 18 (1995). URL http://www.sciencedirect.com/science/article/pii/016763779400060J
8. Labbé, M., Laporte, G., Martello, S.: Upper bounds and algorithms for the maximum cardinality bin packing problem. European Journal of Operational Research **149**(3), 490 – 498 (2003). URL http://www.sciencedirect.com/science/article/pii/S0377221702004666
9. Lodi, A., Martello, S., Vigo, D.: Recent advances on two-dimensional bin packing problems. Discrete Applied Mathematics **123**(1), 379 – 396 (2002). URL http://www.sciencedirect.com/science/article/pii/S0166218X0100347X
10. Martello, S., Toth, P.: Lower bounds and reduction procedures for the bin packing problem. Discrete Applied Mathematics **28**(1), 59 – 70 (1990). URL http://www.sciencedirect.com/science/article/pii/0166218X9090094S
11. Martello, S., Vigo, D.: Exact solution of the two-dimensional finite bin packing problem. Management science **44**(3), 388–399 (1998)
12. Mohr, R., Masini, G.: Good Old Discrete Relaxation. In: Y. Kodratoff (ed.) 8th European Conference on Artificial Intelligence (ECAI '88), pp. 651–656. Pitmann Publishing, Munich, Germany (1988). URL https://hal.inria.fr/inria-00548479
13. Monette, J.-N., Schaus, P., Zampelli, S., Deville, Y., Dupont, P.: A CP approach to the balanced academic curriculum problem. In: Seventh International Workshop on Symmetry and Constraint Satisfaction Problems, vol. 7 (2007)
14. OscaR Team: OscaR: Scala in OR (2012). Available from `https://bitbucket.org/oscarlib/oscar`
15. Pelsser, F., Schaus, P., Régin, J.-C.: Revisiting the cardinality reasoning for binpacking constraint. In: International Conference on Principles and Practice of Constraint Programming, pp. 578–586. Springer (2013)
16. Régin, J., Rezgui, M.: Discussion about constraint programming bin packing models. In: AI for Data Center Management and Cloud Computing, Papers from the 2011 AAAI Workshop, San Francisco, California, USA, August 7, 2011 (2011). URL http://www.aaai.org/ocs/index.php/WS/AAAIW11/paper/view/3817
17. Régin, J.-C.: Generalized arc consistency for global cardinality constraint. In: Proceedings of the thirteenth national conference on Artificial intelligence-Volume 1, pp. 209–215. AAAI Press (1996)
18. Schaus, P.: Solving balancing and bin-packing problems with constraint programming. These de doctorat, Université catholique de Louvain (2009)
19. Schaus, P., Régin, J.-C., Schaeren, R. V., Dullaert, W., Raa, B.: Cardinality reasoning for bin-packing constraint. application to a tank allocation problem. In: International Conference on Principles and Practice of Constraint Programming (2012)
20. Schaus, P., Régin, J.-C., Van Schaeren, R., Dullaert, W., Raa, B.: Cardinality reasoning for bin-packing constraint: application to a tank allocation problem. In: International Conference on Principles and Practice of Constraint Programming, pp. 815–822. Springer (2012)
21. Shaw, P.: A constraint for bin packing. In: International Conference on Principles and Practice of Constraint Programming, pp. 648–662. Springer (2004)
22. Van Cauwelaert, S., Lombardi, M., Schaus, P.: How efficient is a global constraint in practice ? Constraints (2017)