Revisiting the cardinality reasoning for BinPacking constraint

François Pelsser[1], Pierre Schaus[1], Jean-Charles Régin[2]

[1] UCLouvain, ICTEAM,
Place Sainte-Barbe 2,
1348 Louvain-la-Neuve (Belgium),
pierre.schaus@uclouvain.be
[2] University of Nice-Sophia Antipolis,
I3S UMR 6070, CNRS, (France)
jcregin@gmail.com

Abstract. In a previous work, we introduced a filtering for the BinPacking constraint based on a cardinality reasoning for each bin combined with a global cardinality constraint. We improve this filtering with an algorithm providing tighter bounds on the cardinality variables. We experiment it on the Balanced Academic Curriculum Problems demonstrating the benefits of the cardinality reasoning for such bin-packing problems.

Keywords: Constraint Programming, Global Constraints, Bin-Packing

1 Introduction

The `BinPacking`$([X_1, ..., X_n], [w_1, ..., w_n], [L_1, ..., L_m])$ global constraint captures the situation of allocating n indivisible weighted items to m capacitated bins:

- X_i is an integer variable representing the bin where item i, with strictly positive integer weight w_i, is placed. Every item must be placed *i.e.* $Dom(X_i) \subseteq [1..m]$.
- L_j is an integer variable representing the sum of items weights placed into that bin.

The constraint enforces the following relations:

$$\forall j \in [1..m] : \sum_{i|X_i=j} w_i = L_j$$

The initial filtering algorithm proposed for this constraint in [8] essentially filters the domains of the X_i using a knapsack-like reasoning to detect if forcing an item into a particular bin j would make it impossible to reach a load L_j for that bin. This procedure is very efficient but can say that an item is OK for a particular bin while it is not. A failure detection algorithm was also introduced in [8] computing a lower bound on the number of bins necessary to complete the

partial solution. This last consistency check has been extended in [2]. Cambazard and O'Sullivan [1] propose to filter the domains using an LP arc-flow formulation.

In classical bin-packing problems, the capacity of the bins $\overline{L_j}$ are constrained while the lower bounds $\underline{L_j}$ are usually set to 0 in the model. This is why existing filtering algorithms use the upper bounds of the load variables $\overline{L_j}$ (*i.e.* capacity of the bins) and do not focus much on the lower bounds of these variables $\underline{L_j}$.

Recently [7] introduced an additional cardinality based filtering counting the number of items in each bin. We can view this extension as a generalization $\texttt{BinPacking}([X_1, ..., X_n], [w_1, ..., w_n], [L_1, ..., L_m], [C_1, ..., C_m])$ of the constraint where C_j are counting variables, that is defined by $\forall j \in [1..m] : C_j = |\{i | X_i = j\}|$. This formulation for the `BinPacking` constraint is well suited when

- the lower bounds on load variables are also constrained initially $\underline{L_j} > 0$,
- the items to be placed are approximately equivalent in weight (the bin-packing is dominated by an assignment problem), or
- there are cardinality constraints on the number of items in each bin.

The idea of [7] is to introduce a redundant global cardinality constraint [5]:

$$\begin{aligned} \texttt{BinPacking}([X_1, ..., X_n], [w_1, ..., w_n], [L_1, ..., L_m], [C_1, ..., C_m]) \equiv \\ \texttt{BinPacking}([X_1, ..., X_n], [w_1, ..., w_n], [L_1, ..., L_m]) \wedge \\ \texttt{GCC}([X_1, \ldots, X_n], [C_1, \ldots, C_m]) \end{aligned} \quad (1)$$

with a specialized algorithm used to adjust the upper and lower bounds of the C_j variables when the bounds of the L_j's and/or the domains of the X_i's change. Naturally the tighter are the bounds computed on the cardinality variables, the stronger will be the filtering induced by the `GCC` constraint.

We first introduce some definitions, then we recall the greedy algorithm introduced in [7] to update the cardinality variables.

Definition 1. *We denote by $pack_j$ the set of items already packed in bin j : $pack_j = \{i | Dom(X_i) = \{j\}\}$ and by $cand_j$ the candidate items available to go in bin j: $cand_j = \{i | j \in Dom(X_i) \wedge |Dom(X_i)| > 1\}$. The sum of the weights of a set of items S is $sum(S) = \sum_{i \in S} w_i$.*

As explained in [7], a lower bound on the number of items that can be additionally packed into bin j can be obtained by finding the size of the smallest cardinality set $A_j \subseteq cand_j$ such as $sum(A_j) \geq \underline{L_j} - sum(pack_j)$. Then we have $\underline{C_j} \geq |pack_j| + |A_j|$. Thus we can filter the lower bound of the cardinality C_j as follows:

$$\underline{C_j} \leftarrow \max(\underline{C_j}, |pack_j| + |A_j|).$$

This set A_j is obtained in [7] by scanning greedily elements in $cand_j$ with decreasing weights until an accumulated weight of $\underline{L_j} - sum(pack_j)$ is reached. It can be done in linear time assuming the items are sorted initially by weight.

Example 1. Five items with weights $3, 3, 4, 5, 7$ can be placed into bin 1 having a possible load $L_1 \in [20..22]$. Two other items are already packed into that bin with weights 3 and 7 ($|pack_1| = 2$ and $l_1 = 10$). Clearly we have that $|A_1| = 2$ obtained with weights $5, 7$. The minimum value of the domain of the cardinality variable C_1 is thus set to 4.

A similar reasoning can be used to filter the upper bound of the cardinality variable $\overline{C_j}$.

This paper further improves the cardinality based filtering, introducing

1. In Section 2, an algorithm computing tighter lower/upper bounds on the cardinality variables C_j of each bin j, and
2. In Section 3, an algorithm to update the load variables L_j based on the cardinality information.

The new filtering is experimented on the Balanced Academic Curriculum Problem in Section 4.

2 Filtering the cardinality variables

The lower (upper) bound computation on the cardinality C_j introduced in [7] only considers the possible items $cand_j$ and the minimum (maximum) load value to reach *i.e.* $\underline{L_j}$ ($\overline{L_j}$). Stronger bounds can possibly be computed by also considering the cardinality variables of other bins. Indeed, an item which is used for reaching the minimum cardinality or minimum load for a bin j, may not be usable again for computing the minimum cardinality of another bin k as illustrated on next example:

Example 2. A bin j can accept items having weights $3, 3, 3$ with a minimum load of 6 and thus a minimum cardinality of 2 items. A bin k with a minimum load of 5 can accept the same items plus two items of weight 1. Clearly, the bin k can not take more than one item with weight 3 for computing its minimum cardinality because it would prevent the bin j to reach its minimum cardinality of 2. Thus the minimum cardinality of bin k should be 3 and not 2 as would be computed with the lower bound of [7].

Minimum Cardinality of bin j Algorithm 1 computes a stronger lower bound also taking into account the cardinality variables of other bins $\underline{C_k}\ \forall k \neq j$. The intuition is that it prevents to reuse again an item if it is required for reaching a minimum cardinality in another bin. This is achieved by maintaining for every other bin k the number of items this bin is ready to give without preventing it to fulfill its own minimum cardinality requirement $\underline{C_k}$.

Clearly if a bin k must pack at least $\underline{C_k}$ items and has already packed $|pack_k|$ items, this bin can not give more than $|cand_k| - (\underline{C_k} - |pack_k|)$ items to bin j. This information is maintained into the variables $availableForOtherBins_k$ initialized at line 5.

Example 3. Continuing on Example 2, bin j will have $availableForOtherBins_j = 3 - (2 - 0) = 1$ because this bin can give at most one of its item to another bin.

Since items are iterated in decreasing weight order at line 7, the other bins accept to give first their "heaviest" candidate items. This is an optimistic situation from the point of view of bin j, justifying why the algorithm computes a valid lower bound on the cardinality variable C_j. Each time an item is used by bin j, the other bins (where this item was candidate) reduce their quantities $availableForOtherBins_k$ since they "consume" their flexibility to give items. If at least one other bin k absolutely needs the current item i to fulfill its own minimum cardinality (detected at line 13), `available` is set to `false` meaning that this item can not be used in the computation of the cardinality of bin j to reach the minimum load.

On the other hand, if the current item can be used (`available=true`), then other bins which agreed to give this item have one item less available. The $availableForOtherBins_k$ numbers are decremented at line 22.

Finally notice that the algorithm may detect unfeasible situations when it is not able to reach the minimum load at line 28.

Maximum Cardinality The algorithm to compute the maximum cardinality is similar. The changes to bring to Algorithm 1 are:

1. The variable $binMinCard$ should be named $binMaxCard$
2. The items are considered in increasing weight order at line 7, and
3. The stopping criteria at line 8 becomes $binLoad + w_i > \overline{L_j}$.
4. There is no feasibility test at lines 27 - 29.

Complexity Assuming the items are sorted initially in decreasing weights, this algorithm runs in $O(n \cdot m)$ with n the number of items and m the number of bins. Hence adjusting the cardinality of every bins takes $O(n \cdot m^2)$. This algorithm has no guarantee to be idempotent. Indeed the bin j may consider an item i as available, but the later adjustment of the minimum cardinality of another bin k may cause this item to be unavailable if bin j is considered again.

Example 4. The instance considered - depicted in Figure 1 (a) - is the following:

$$\begin{aligned} &\texttt{BinPacking}([X_1, \ldots, X_4], [w_1, \ldots, w_4], [L_1, \ldots, L_3]) \\ &X_1 \in \{1,2\}, X_2 \in \{1,2\}, X_3 \in \{2,3\}, X_4 \in \{2,3\}, \\ &w_1 = 1, w_2 = 1, w_3 = 3, w_4 = 3 \\ &L_1 \in \{1,2\}, L_2 \in \{2,3\}, L_3 \in \{2,4\} \end{aligned} \tag{2}$$

We consider first the computation of the cardinality of bin 2. This bin must have at least one item to reach its minimum load. We now consider the maximum cardinality of this bin. Items 1 and 2 can both be packed into bin 2 but doing

Algorithm 1: Computes a lower bound on the cardinality of bin j

Data: j a bin index
Result: $binMinCard$ a lower bound on the min cardinality for the bin j

$binLoad \leftarrow sum(pack_j)$;
$binMinCard \leftarrow |pack_j|$;
$othersBins \leftarrow \{1, \ldots, m\} \setminus j$;
foreach $k \in otherBins$ **do**
  $availableForOtherBins_k \leftarrow |cand_k| - (\underline{C_k} - |pack_k|)$;
end
foreach $i \in cand_j$ *in decreasing weight order* **do**
  if $binLoad \geq \underline{L_j}$ **then**
    break ;
  end
  available $\leftarrow$ true;
  for $k \in othersBins$ **do**
    if $k \in Dom(X_i) \ \wedge availableForOtherBins_k = 0$ **then**
      available $\leftarrow$ false ;
    end
  end
  if *available* **then**
    $binLoad \leftarrow binLoad + w_i$;
    $binMinCard \leftarrow binMinCard + 1$;
    for $k \in othersBins$ **do**
      if $k \in Dom(X_i)$ **then**
        $availableForOtherBins_k \leftarrow availableForOtherBins_k - 1$;
      end
    end
  end
end
if $binLoad < \underline{L_j}$ **then**
  The constraint is unfeasible ;
end

so would prevent bin 1 to achieve its minimum load requirement of 1. Hence only one of these items can be used during the computation of the maximum cardinality for bin 2. Assuming that item 1 is used, the next item considered is item 3 having a weight of 3. But Adding this item together with item 1 would exceed the maximum load ($4 > 3$) (stopping criteria for the maximum cardinality computation). Hence the final maximum cardinality for bin 2 is one. The cardinality reasoning also deduces that bin 1 must have between one and two items and bin 3 must have exactly one item. Based on these cardinalities, the global cardinality constraint (`GCC`) is able to deduce that item 1 and 2 must be packed into bin 1. This filtering is illustrated on Figure 1 (b).

The algorithm from [7] deduces that bin 2 must have between one and two items (not exactly one as the new filtering). The upper bound of two items is obtained with the two lightest items 1 and 2. As for the new algorithm, it deduces

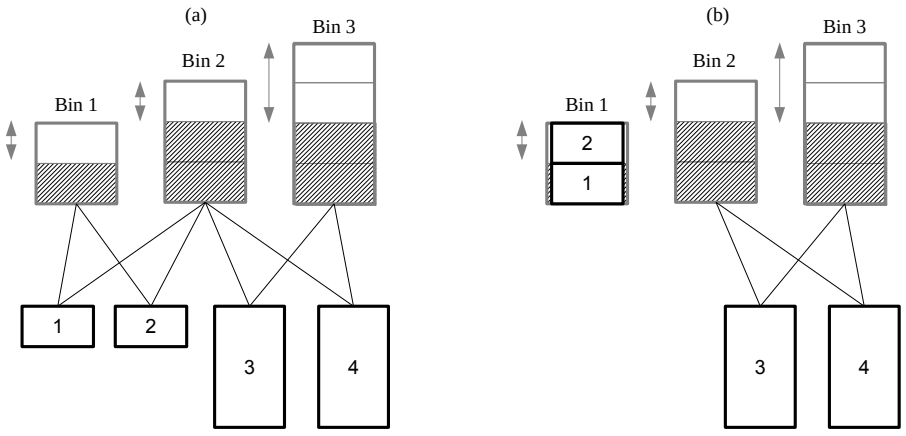

Fig. 1: (a) BinPacking instance with 3 bins and 4 items. The arcs represent for each item, the possible bins. (b) Domains resulting from the filtering induced with the tighter computation of the cardinalities. The grey in a bin stands for the minimum level to reach.

that bin 1 must have between one and two items and bin 3 must have exactly one item. Unfortunately, the `GCC` is not able to remove any bin from the item's domains based on these cardinality bounds. Thus, this algorithm is less powerful than the new one.

3 Filtering the load variables

We introduce a filtering of the load variable taking the cardinality information into account. No such filtering was proposed in [7]. Algorithm 2 is similar to Algorithm 1 except that we try to reach the minimum cardinality requirements by choosing first the "lightest" items until the minimum cardinality $\underline{C_j}$ is reached (line 8). Again a similar reasoning can be done to compute an upper bound on the maximum load.

4 Experiments

The Balanced Academic Curriculum Problem (BACP) is recurrent in Universities. The goal is to schedule the courses that a student must follow in order to respect the prerequisite constraints between courses and to balance as much as possible the workload of each period. Each period also has a minimum and maximum number of courses. The largest of the three instances available on CSPLIB (http://www.csplib.org) with 12 periods, 66 courses having a weight

Algorithm 2: Computes a lower bound on load of bin j

Data: j a bin index
Result: $binMinLoad$ a lower bound on the load of bin j

$binCard \leftarrow |pack_j|$;
$binMinLoad \leftarrow sum(pack_j)$;
$othersBins \leftarrow \{1, \ldots, m\} \setminus j$;
foreach $k \in otherBins$ **do**
  $availableForOtherBins_k \leftarrow |cand_k| - (\underline{C_k} - |pack_k|)$;
end
foreach $i \in cand_j$ *in increasing weight order* **do**
  if $binCard \geq \underline{(C_j)}$ **then**
    break ;
  end
  available $\leftarrow$ true;
  for $k \in othersBins$ **do**
    if $k \in Dom(X_i) \wedge availableForOtherBins_k = 0$ **then**
      available $\leftarrow$ false ;
    end
  end
  if *available* **then**
    $binMinLoad \leftarrow binLoad + w_i$;
    $binCard \leftarrow binCard + 1$;
    for $k \in othersBins$ **do**
      if $k \in Dom(X_i)$ **then**
        $availableForOtherBins_k \leftarrow availableForOtherBins_k - 1$;
      end
    end
  end
end
if $binCard < \underline{C_j}$ **then**
  The constraint is unfeasible ;
end

limit(s)	A	B	C
15	13	27	**41**
30	18	34	**46**
60	21	37	**51**
120	25	43	**57**
1800	37	62	**69**

Table 1: Number of instances for which is was possible to prove optimality within the time limit.

between 1 and 5 (credits) and 65 prerequisites relations, was modified in [6] to generate 100 new instances[3] by giving each course a random weight between

[3] Available at http://becool.info.ucl.ac.be/resources/bacp

1 and 5 and by randomly keeping 50 out of the 65 prerequisites. Each period must have between 5 and 7 courses. As shown in [3], a better balance property is obtained by minimizing the variance instead of the maximum load. For each instance, we test three different filtering configurations for bin-packing:

- A: The `BinPacking` constraint from [8] + a `GCC` constraint,
- B: A + the cardinality filtering from [7],
- C: A + the cardinality filtering introduced in this paper.

	time (ms)			best bound			number of failures		
instance	A	B	C	A	B	C	A	B	C
inst2.txt	timeout	timeout	**679**	3243	3247	**3237**	835459	1064862	**829**
inst14.txt	timeout	45625	**6925**	3107	**3105**	**3105**	1043251	228294	**8530**
inst22.txt	timeout	13971	**281**	3045	**3041**	**3041**	811852	48482	**353**
inst30.txt	timeout	118964	**192**	3416	**3402**	**3402**	795913	707487	**129**
inst36.txt	timeout	timeout	**337**	2685	2685	**2671**	847641	915849	**364**
inst47.txt	timeout	timeout	**112**	3309	3309	**3303**	2561038	3812512	**269**
inst65.txt	timeout	timeout	**222**	3416	3414	**3402**	921694	1091396	**168**
inst70.txt	timeout	timeout	**101060**	3043	3043	**3041**	1917729	1516627	**125270**
inst87.txt	16275	15089	**251**	**3643**	**3643**	**3643**	109173	65493	**207**
inst98.txt	timeout	timeout	**48**	2987	2987	**2979**	7023383	8261509	**261**

Table 2: Detailed statistics obtained on some significant instances.

The experiments were conducted on a Macbook Pro 2.3 Ghz, I7. The solver used is OscaR [4] running on JVM 1.7 of Oracle and implemented with Scala 2.10. The source code of the constraint is available on OscaR repository.

Table 1 gives the number of solved instances for increasing timeout values. Table 2 illustrates the detailed numbers (time, best bound, number of failures) for some instances with a 30 minutes timeout. As can be seen, the new filtering allows to solve more instances sometimes cutting the number of failures by several order of magnitudes.

5 Conclusion

We introduced stronger cardinality bounds on the `BinPacking` constraint by also integrating the cardinality requirements of other bins during the computation. These stronger bounds have a direct impact on the filtering of placement variables through the `GCC` constraint. The improved filtering was experimented on the BACP allowing to solve more instances and reducing drastically the number of failures on some instances.

References

1. Hadrien Cambazard and Barry O'Sullivan. Propagating the bin packing constraint using linear programming. In David Cohen, editor, *CP*, volume 6308 of *Lecture Notes in Computer Science*, pages 129–136. Springer, 2010.
2. Julien Dupuis, Pierre Schaus, and Yves Deville. Consistency check for the bin packing constraint revisited. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *CPAIOR*, volume 6140 of *Lecture Notes in Computer Science*, pages 117–122. Springer, 2010.
3. Jean-Noël Monette, Pierre Schaus, Stéphane Zampelli, Yves Deville, and Pierre Dupont. A CP approach to the balanced academic curriculum problem. In *Seventh International Workshop on Symmetry and Constraint Satisfaction Problems*, volume 7, 2007.
4. OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.
5. Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the thirteenth national conference on Artificial intelligence-Volume 1*, pages 209–215. AAAI Press, 1996.
6. Pierre Schaus et al. *Solving balancing and bin-packing problems with constraint programming*. PhD thesis, PhD thesis, Universit catholique de Louvain Louvain-la-Neuve, 2009.
7. Pierre Schaus, Jean-Charles Régin, Rowan Van Schaeren, Wout Dullaert, and Birger Raa. Cardinality reasoning for bin-packing constraint: Application to a tank allocation problem. In Michela Milano, editor, *CP*, volume 7514 of *Lecture Notes in Computer Science*, pages 815–822. Springer, 2012.
8. Paul Shaw. A constraint for bin packing. In *Principles and Practice of Constraint Programming–CP 2004*, pages 648–662. Springer, 2004.