Enhanced Pseudo-Polynomial Formulations for Bin Packing and Cutting Stock Problems

Maxence Delorme(1), Manuel Iori(2)

(1) DEI "Guglielmo Marconi", University of Bologna
(2) DISMI, University of Modena and Reggio Emilia

October 13, 2017

Abstract

We study pseudo-polynomial formulations for the classical bin packing and cutting stock problems. We first propose an overview of dominance and equivalence relations among the main pattern-based and pseudo-polynomial formulations from the literature. We then introduce reflect, a new formulation that uses just half of the bin capacity to model an instance and needs significantly less constraints and variables than the classical models. We propose upper and lower bounding techniques that make use of column generation and dual information to compensate reflect weaknesses when bin capacity is too high. We also present non-trivial adaptations of our techniques that solve two interesting problem variants, namely, the variable sized bin packing problem and the bin packing problem with item fragmentation. Extensive computational tests on benchmark instances show that our algorithms achieve state of the art results on all problems, improving upon previous algorithms and finding several new proven optimal solutions.

Keywords: bin packing, cutting stock, pseudo-polynomial, equivalent models, variable size, fragmentation.

1 Introduction

The *bin packing problem* (BPP) requires to pack a set of weighted items into the minimum number of identical capacitated bins. The *cutting stock problem* (CSP) is the BPP version in which all items having the same weight are grouped together into item types. The term packing is normally adopted for applications devoted to the minimization of the number of boxes (containers, cells, ...) required to allocate a set of items, and the term cutting for industrial process where stocks of a material (steel bars, pipes, ...) have to be cut into demanded items while minimizing waste. Apart from these applications, the BPP and the CSP also serve in the optimization of a variety of real-world problems, including capacitated vehicle routing, resource-constrained scheduling problems, and production systems, just to name a few. Their wide range of applications motivated a large research interest, and, indeed, the two problems have been tackled by almost all known combinatorial optimization techniques. We refer to the recent surveys by Coffman et al. (2013), who focused on approximation algorithms, by Alves et al. (2016), who studied the use of dual-feasible functions, and by Delorme et al. (2016), who reviewed and tested exact algorithms and mathematical models.

The BPP and the CSP have also been the training ground of some *mixed integer linear programming* (MILP) models, and corresponding solution algorithms, that later became useful tools for many other combinatorial optimization problems. Notably, Gilmore and Gomory (1961) modeled the CSP as a set-covering by using a pattern-based representation, and then proposed the well-known column generation algorithm. The strength of their model comes from the very high quality of its continuous relaxation value. Such relaxation (and other variants discussed in Section 3) are at the basis of the most effective branch-and-price algorithms for the solution of the BPP and the CSP (see, e.g., Vanderbeck 1999 and Belov and Scheithauer 2006), and of many generalizations of practical interest (see, e.g., Arbib et al. 2002 and Ceselli and Righini 2008).

The number of variables in the cited pattern-based models is exponential in the number of items. A different branch of the literature focused, instead, on modeling the BPP and the CSP with *pseudo-polynomial* MILP models, that is, formulations in which the numbers of variables and constraints are polynomials in both the number of items and the bin capacity. Already in the 1960s, Shapiro (1968) showed the connection that exists between integer programming and *dynamic programming* (DP), by modeling the knapsack problem as a shortest path on a network of pseudo-polynomial size, where (i) nodes are associated with different levels of occupation of the knapsack, (ii) items are associated with arcs having length equal to the item weight, and (iii) decision variables are associated with arcs. Wolsey (1977) extended this idea to the case of problems with multiple knapsack inequalities, solving the CSP as a network flow with side constraints. The research was later continued by Valério de Carvalho (1999), who proposed a CSP MILP model, called *arc-flow*, and showed that its continuous relaxation value is equal to that of Gilmore and Gomory (1961). A related MILP model, known as *one-cut*, was independently developed by Rao (1976) and Dyckhoff (1981). In contrast with arc-flow, one-cut solves the CSP by using variables that represent the physical positions of the cuts. Items are obtained by performing the selected cuts one at a time, either along the bin or along residual bin portions obtained by previous cuts.

The mentioned research on pseudo-polynomial formulations had mainly a theoretical interest, because the large size of these models made them unsolvable in practice even for medium instances. In recent years, however, the sharp development of MILP commercial software and the increase in computational power made these formulations viable tools to solve the BPP, the CSP and several other *cutting and packing* (C&P) problems. Among others, Brandão and Pedroso (2016) obtained good results on one-dimensional C&P problems by improving arc-flow, Furini et al. (2016) generalized one-cut to solve some two-dimensional guillotine cutting problems, and Côté et al. (2014) and Delorme et al. (2017b) used pseudo-polynomial models as stepping stones of decomposition algorithms for two-dimensional packing problems. This research line has a great potential interest, because pseudo-polynomial models are fairly easy to implement, thus representing a useful tool for practitioners, and appear not only in C&P but in many other fields, including vehicle routing and scheduling (see, e.g., Macedo et al. 2011 and Pessoa et al. 2010) just to cite some.

In this paper, we continue the research on pseudo-polynomial models presenting results of both theoretical and computational interest. We first focus on the relation that exists among well-known pattern-based and pseudo-polynomial formulations, providing for the first time a complete picture of equivalences and dominances among them. Then, we present a new effective formulation, called *reflect*, that requires only half of the bin capacity to model a CSP instance. As happens for all pseudo-polynomial formulations, also reflect has troubles in solving instances with very large capacities. We thus improve it with several techniques, leading to an algorithm, called *reflect+*, that has an outstanding computational performance. In detail, we provide the following contributions:

- We prove that one-cut and arc-flow formulations for the CSP are equivalent (i.e., they have the same continuous relaxation value), closing a long-standing research gap;
- We extend the previous result and provide a clear picture of the dominance and equivalence relations that exist among the main pattern-based and pseudo-polynomial MILP formulations

that have been proposed for the BPP and the CSP;

- We introduce our new formulation, reflect, and show that it improves the classical arc-flow by needing significantly less constraints and variables;
- We develop improvement techniques based on the combined use of reflect with column generation, dual cuts, and heuristics. We show that our heuristics are faster and more efficient than traditional approaches based on the solution of set-covering models with restricted sets of columns;
- We extend reflect and reflect+ by devising formulations and algorithms that solve two important BPP variants, namely, the variable-sized BPP and the BPP with item fragmentation;
- We perform extensive computational tests and show that our algorithms achieve state of the art results, improving previous algorithms from the literature, finding several new optimal solutions for the BPP and the CSP, and solving to optimality all attempted instances of the variable-sized BPP and the BPP with item fragmentation.

The remainder of the paper is organized as follows. In Section 2, we give the necessary notation and review the main formulations proposed for the BPP and the CSP. In Section 3, we establish the full set of relations among the existing formulations. In Section 4, we present reflect and reflect+. The variable-sized BPP is solved in Section 5, and the BPP with item fragmentation in Section 6. In Section 7, we present the outcome of extensive computational experiments, and then, in Section 8, we draw some conclusions. For the sake of conciseness, all proofs of our statements and some additional algorithms and examples are reported in an *appendix*.

2 The BPP, the CSP, and their well-known MILP formulations

In this section, we formally describe the BPP and the CSP, give the necessary notation, and present the main formulations developed in the literature for their solution.

2.1 Problem description and notation

In the BPP, we are given a set of n items, each having weight w_j ($j = 1, \dots, n$), and an unlimited supply of identical bins of capacity c. The aim is to pack all items into the minimum number of bins, so that the sum of the item weights in any bin does not exceed the capacity. To better adapt to either the concept of packing or that of cutting, in the following we use alternatively the terms *weight* or *width* when referring to w_j. We assume that all input values are integer and $0 < w_j < c$ holds for any j. The CSP has the same aim of the BPP, but, apart from the unlimited supply of stocks (bins) of capacity c, its input consists of a set of m item types. Each item type j has a demand of d_j items, all having width w_j ($j = 1, \dots, m$). A CSP instance can be obtained from a BPP one by grouping into item types all items having the same width. So, a solution method for the CSP also solves the BPP, and viceversa. Unless stated otherwise, the formulations that we report below refer to the CSP.

We use F_X to denote a generic MILP formulation. We use $L(F_X)$ to denote the continuous (linear programming) relaxation of F_X, which is obtained by dropping the integrality constraints from F_X. When no confusion arises, we also use $L(F_X)$ to denote the optimal solution value of $L(F_X)$. We say that a formulation F_X is *equivalent* to a formulation F_Y, if $L(F_X) = L(F_Y)$ holds for any instance. As we deal with minimization problems, we say that a formulation F_X *dominates* a formulation F_Y, if $L(F_X) \geq L(F_Y)$ holds for any instance and $L(F_X) > L(F_Y)$ holds for at least one instance. If F_X dominates F_Y, then we also say that F_X *is included* into F_Y. Suppose a given formulation contains a variable x: We use the notation $\bar{x}$ to denote the value taken by x in a given solution of that formulation.

2.2 Pattern-based formulations

We define a pattern p as an array $(a_{1p}, \dots, a_{mp})$, with a_{jp} being a non-negative integer that gives the number of items of type j that are included in the pattern. Let P define the class of feasible patterns, i.e., those patterns p for which $\sum_{j=1}^{m} a_{jp} w_j \leq c$ holds. Gilmore and Gomory (1961) associated with each pattern p an integer decision variable ξ_p, indicating the number of times the pattern is selected, and modeled the CSP as the following set-covering problem

$$(F_{GG}) \quad \left\{ \min \ z = \sum\nolimits_{p \in P} \xi_p : \ \sum\nolimits_{p \in P} a_{jp} \xi_p \geq d_j \text{ for } j = 1, \dots, m, \ \xi_p \in \mathbb{N} \text{ for } p \in P \right\}. \tag{1}$$

Note that in (1) nothing prevents a_{jp} from being larger than d_j, and hence there could be optimal solutions of F_{GG}, or of its continuous relaxation $L(F_{GG})$, that include patterns containing a number of items larger than their demands. Consider the following example.

Example 1 *A CSP instance with $m = 3$, $w = (7, 4, 3)$, $d = (1, 1, 1)$, and $c = 11$.*

An optimal $L(F_{GG})$ solution of Example 1 has value 4/3 and consists of selecting three patterns: $(1, 1, 0)$ (i.e., a pattern containing a copy of the item of width 7 and a copy of the item of width 4) with $\bar{\xi}_1 = 2/3$; $(1, 0, 1)$ with $\bar{\xi}_2 = 1/3$; and $(0, 1, 2)$ with $\bar{\xi}_3 = 1/3$.

The *proper relaxation* (see, e.g., Nitsche et al. 1999) overcomes this F_{GG} drawback by focusing on a restricted set P' of feasible patterns. Each pattern $p \in P'$ satisfies $\sum_{j=1}^{m} a_{jp} w_j \leq c$ and also $0 \leq a_{jp} \leq d_j$ for $j = 1, \dots, m$. The CSP can then be modeled as

$$(F_{PR}) \quad \left\{ \min \ z = \sum\nolimits_{p \in P'} \xi_p : \ \sum\nolimits_{p \in P'} a_{jp} \xi_p \geq d_j \text{ for } j = 1, \dots, m, \ \xi_p \in \mathbb{N} \text{ for } p \in P' \right\}. \tag{2}$$

Because $P' \subseteq P$, we can state that $L(F_{PR}) \geq L(F_{GG})$. Consider now Example 1. An optimal solution of $L(F_{PR})$ has value 3/2 and consists of selecting the three following patterns: $(1, 1, 0)$ with $\bar{\xi}_1 = 1/2$; $(1, 0, 1)$ with $\bar{\xi}_2 = 1/2$; and $(0, 1, 1)$ with $\bar{\xi}_3 = 1/2$. We can thus conclude with the (known) fact that F_{PR} dominates F_{GG}.

Note that for most of the CSP instances the rounded-up lower bound values of the two formulations ($\lceil L(F_{GG}) \rceil$ and $\lceil L(F_{PR}) \rceil$) coincide. Still, there are cases for which $\lceil L(F_{GG}) \rceil < \lceil L(F_{PR}) \rceil$. Such cases have been investigated intensively in studies over the *Mixed-Integer Round-Up* (MIRUP) conjecture, which states that both $(z_{opt} - \lceil L(F_{GG}) \rceil) \leq 1$ and $(z_{opt} - \lceil L(F_{PR}) \rceil) \leq 1$ hold for any CSP instance, with z_{opt} being the optimal integer solution value. For further details, we refer to, e.g., Caprara et al. (2015) and Kartak et al. (2015). Note also that branch-and-price algorithms explicitly devoted to the solution of the BPP are based on patterns with binary a_{jp} values, and hence they use the proper relaxation.

2.3 Pseudo-polynomial formulations

Pseudo-polynomial formulations involve a large number of variables and constraints, and hence they have been usually proposed in the literature together with appropriate reduction techniques. We focus in this section on basic models that do not make use of any reduction, and discuss existing reduction techniques at the end of Section 3.

The *one-cut formulation* (F_{OC}) was formally introduced by Rao (1976) and Dyckhoff (1981). It simulates the physical cutting process by choosing a set of cuts, each dividing the bin (or a portion of the bin) into two smaller pieces, a left one and a right one. While the left piece is an item, the right piece is either an item or a residual that can be re-used to produce smaller items with successive cuts. To describe the model, we need some additional notation. Let $\mathcal{W} = \{w_1, w_2, \dots\}$ define the set of different item widths, and $\mathcal{S}$ be the set of item widths combinations not exceeding c. Set $\mathcal{S}$ can be computed trough a standard DP and can be formally stated as

$$\mathcal{S} = \left\{ \bar{w} = \sum\nolimits_{j=1}^{m} w_j x_j,\ \bar{w} \leq c,\ x_j \in \mathbb{N} \text{ for } j = 1, \ldots, m \right\}. \tag{3}$$

Let $\mathcal{R}$ define the set of residual widths, computed as $\mathcal{R} = \{c - \bar{w} : \bar{w} \in \mathcal{S} \text{ and } \bar{w} \leq c - \min_j\{w_j\}\}$. The width of any left piece produced by a cut is in $\mathcal{W}$, while the width of any right piece (including c) is in $\mathcal{R}$. For a given width $q \in \mathcal{W} \cup \mathcal{R}$, let L_q give the demand of q, i.e., $L_q = d_j$ if $\exists\ j$ having $w_j = q$ and $L_q = 0$ otherwise. One-cut makes use of three additional sets of widths. Set $A(q)$ contains piece widths that can be used for producing a left piece of width q, if any: $A(q) = \{p \in \mathcal{R} : p > q\}$ for $q \in \mathcal{W}$, and $A(q) = \emptyset$ for $q \notin \mathcal{W}$. Set $B(q) = \{p \in \mathcal{W} : p + q \in \mathcal{R}\}$ contains item widths that, whether cut as a left piece, would produce a right piece of width q, for $q \in \mathcal{R}$. Set $C(q) = \{p \in \mathcal{W} : p < q\}$ contains item widths that can be cut as a left piece by using a residual of width q, for $q \in \mathcal{R}$.

One-cut uses an integer decision variable y_{pq} giving the number of pieces of width p that are cut into a left piece of width q. The model is then

$$(F_{OC}) \quad \min \quad z = \sum_{q \in \mathcal{W}} y_{cq} \tag{4}$$

$$\text{s.t.} \quad \sum_{p \in A(q)} y_{pq} + \sum_{p \in B(q)} y_{p+q,p} \geq L_q + \sum_{r \in C(q)} y_{qr} \qquad q \in \mathcal{W} \cup \mathcal{R} \setminus \{c\} \tag{5}$$

$$y_{pq} \in \mathbb{N} \qquad p \in \mathcal{R}, q \in \mathcal{W}, p > q. \tag{6}$$

While function (4) minimizes the number of bins used, constraints (5) ensure that the sum of the left and right pieces having width q is not smaller than the demand of width q plus the number of times a residual of width q is re-cut to produce other items.

The *arc-flow formulation* (F_{AF}) was formally proposed for the CSP by Valério de Carvalho (1999) and builds upon the networks in Shapiro (1968) and Wolsey (1977). Formally, let $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ be a graph having vertex set $\mathcal{V} = \{0, 1, \ldots, c-1, c\}$ and arc set $\mathcal{A} = \mathcal{A}_{\text{I}} \cup \mathcal{A}_{\ell}$, where $\mathcal{A}_{\text{I}} = \{(d, e) : d, e \in \mathcal{S} \text{ and } \exists\ j \in \{1, \ldots, m\} : e - d = w_j\}$ is the set of *item arcs*, and $\mathcal{A}_{\ell} = \{(d, d+1) : d = 0, 1, \ldots, c-1\}$ is the set of *loss arcs*. Items arcs model the items' positions in the bin, accomplishing with (3), whereas loss arcs model empty bin portions. The filling of a bin corresponds to a path from root node 0 to sink node c. Let $\delta^+(e)$ (respectively, $\delta^-(e)$) give the subset of arcs emanating from (respectively, entering) vertex e. By introducing an integer variable x_{de} equal to the number of times arc (d, e) is chosen, the CSP becomes

$$(F_{AF}) \quad \min \quad z \tag{7}$$

$$\text{s.t.} \quad \sum_{(e,f) \in \delta^+(e)} x_{ef} - \sum_{(d,e) \in \delta^-(e)} x_{de} = \begin{cases} z & \text{if } e = 0 \\ -z & \text{if } e = c \\ 0 & \text{for } e = 1, \ldots, c-1 \end{cases} \tag{8}$$

$$\sum_{(d,d+w_j) \in \mathcal{A}} x_{d,d+w_j} \geq d_j \qquad j = 1, \ldots, m \tag{9}$$

$$x_{de} \in \mathbb{N} \qquad (d, e) \in \mathcal{A}. \tag{10}$$

Objective function (7) minimizes the number of bins, expressed by variable z, whereas constraints (8) impose flow conservation and constraints (9) ensure that demands are fulfilled.

The *dynamic programming-flow formulation* (F_{DP}) was formally introduced for the BPP by Cambazard and O'Sullivan (2010), but, similarly to F_{AF}, has origins in the early works by Shapiro (1968) and Wolsey (1977). It can be seen as a disaggregated form of F_{AF} that uses an expanded graph obtained from a DP table. In detail, let $\mathcal{G}' = (\mathcal{V}', \mathcal{A}')$ be a DP graph, where $\mathcal{V}' = \{(j, d) : j = 0, \ldots, n; d = 0, \ldots, c\} \cup \{(n+1, c)\}$ has a vertex for each DP state plus a dummy node $(n+1, c)$, and $\mathcal{A}' = \{((j, d), (j+1, e)) : (j, d) \in \mathcal{V}'; (j+1, e) \in \mathcal{V}'\}$ contains arcs connecting two consecutive DP states. There are two types of arcs: Those for which $e = d + w_j$ model the selection of item j when the bin is partially filled with d units; those having $e = d$ discard instead the selection of j in d. All vertices (j, d) having $j = n$ are connected to the dummy vertex $(n+1, c)$. A path from $(0, 0)$ to $(n+1, c)$ represents a feasible bin filling. A decision variable $\varphi_{j,d,j+1,e}$ is associated with each arc $((j, d), (j+1, e)) \in \mathcal{A}'$. By setting $\mathcal{V}'_0 = \mathcal{V}' \setminus \{(0, 0), (n+1, c)\}$, the BPP can be modeled as

$$(F_{DP}) \quad \min \quad z \tag{11}$$

$$\text{s.t.} \quad \sum_{((j,d),(j+1,e)) \in \delta^+((j,d))} \varphi_{j,d,j+1,e} - \sum_{((j-1,e),(j,d)) \in \delta^-((j,d))} \varphi_{j-1,e,j,d} = \begin{cases} z & \text{if } (j,d) = (0,0) \\ -z & \text{if } (j,d) = (n+1,c) \\ 0 & \text{if } (j,d) \in \mathcal{V}'_0 \end{cases} \tag{12}$$

$$\sum_{((j-1,d),(j,d+w_j)) \in \mathcal{A}} \varphi_{j-1,d,j,d+w_j} = 1 \qquad j = 1, \ldots, n \tag{13}$$

$$\varphi_{j,d,j+1,e} \in \mathbb{N} \qquad ((j, d), (j+1, e)) \in \mathcal{A}'. \tag{14}$$

Constraints (12) force flow conservation and constraints (13) impose that each item, adopting the BPP notation, is selected once. Variables are set to take integer values by (14), but, due to (13), variables associated with $((j-1, d), (j, d + w_j))$ arcs are binary. A CSP instance can be modeled by F_{DP} by splitting each item type j into d_j items, obtaining a BPP instance with $n = \sum_{i=1}^{m} d_j$ items.

3 Relations among models

In this section, we prove the relations that exist among the introduced patterns-based and pseudo-polynomial formulations. To the best of our knowledge, this is the first time that a complete characterization of this area of research is provided. We first need two preliminary results.

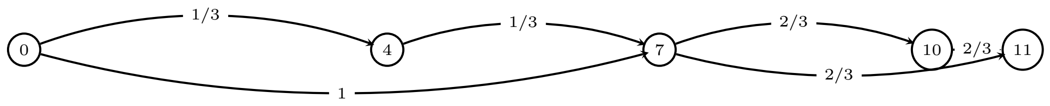
Lemma 1 *(Valério de Carvalho 1999) Any solution of arc-flow or of its continuous relaxation can be decomposed into a set of paths.*

Lemma 1 is based on the decomposition of non-negative flows into paths and cycles (see Ahuja et al. 1993, Chapter 3), with the only remark that, being the arc-flow graph acyclic, only paths may occur. Consider Example 1: An optimal $L(F_{AF})$ solution, shown in Figure 1-(a), has value 4/3 and consists of $\bar{x}_{0,7} = 1$, $\bar{x}_{0,4} = 1/3$, $\bar{x}_{4,7} = 1/3$, $\bar{x}_{7,10} = 2/3$, $\bar{x}_{7,11} = 2/3$, and $\bar{x}_{10,11} = 2/3$. By applying to this solution the procedure given in Section Appendix A, one obtains the decomposed flow made by three paths which is depicted in Figure 1-(b).

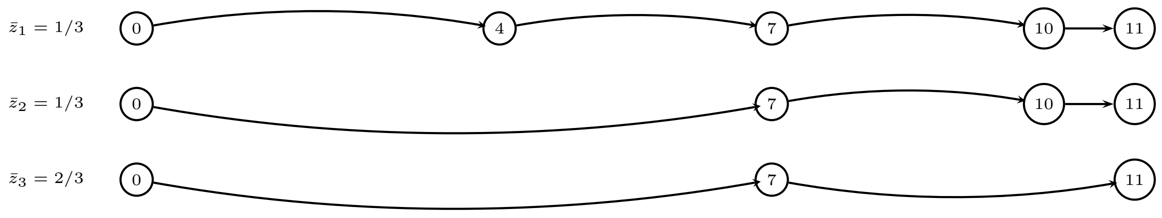
Lemma 2 *Any solution of one-cut or of its continuous relaxation can be decomposed into a set of binary trees.*

Proof. Given in Section Appendix B. □

Consider again Example 1: An optimal $L(F_{OC})$ solution has value 4/3 and consists of $\bar{y}_{11,7} = 1$, $\bar{y}_{11,4} = 1/3$, $\bar{y}_{7,3} = 1/3$, and $\bar{y}_{4,3} = 2/3$. By applying the procedure described in Section Appendix B, we obtain the three trees depicted in Figure 2. The leaves of each tree correspond to either produced items (as leaves 4, 3, and 3 in the left-most tree) or residuals (as leaf 1 in the left-most tree).



(a) An optimal $L(F_{AF})$ solution of value 4/3



(b) Decomposition of the $L(F_{AF})$ solution above into paths ($\bar{z}_p$ = flow on path p)

Figure 1: $L(F_{AF})$ solution and path decomposition for Example 1

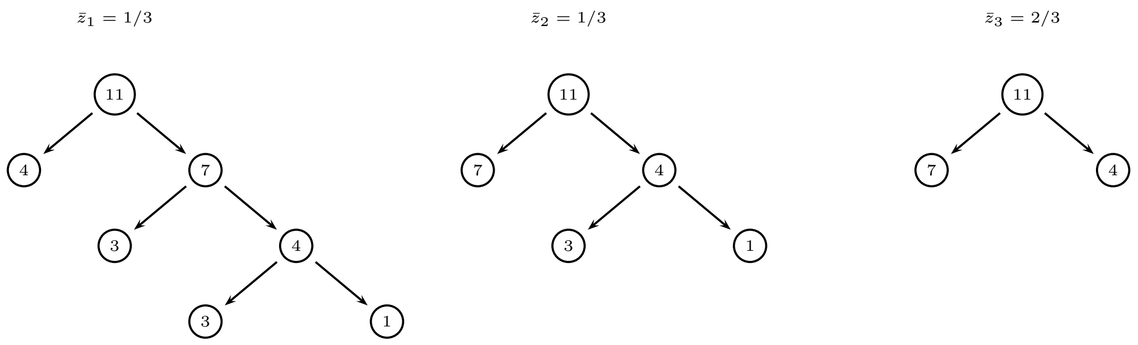


Figure 2: An optimal $L(F_{OC})$ solution of Example 1 decomposed into a set of trees ($\bar{z}_t$ = value of tree t)

Intuitively, we can state a relation between the paths of the decomposed arc-flow solution and the trees of the decomposed one-cut solution. Figures 1 and 2 show a well-constructed example of this relation (consider one at a time paths from top to bottom and trees from left to right). In reality, a few cases should be considered, especially in consideration of the fact that solutions of arc-flow and one-cut are not guaranteed to be left-aligned (as happens in the figures). We leave the technicalities to the appendix, and provide our first result in terms of models relations.

PROPOSITION 1 *F_{AF} is equivalent to F_{OC}.*

Proof. Given in Section Appendix C. □

Note that the proof of Proposition 1 contains two algorithms to directly transform an F_{AF} solution into an F_{OC} one, and viceversa. Note also that, by using the equivalence between F_{AF} and F_{GG} proved in Valério de Carvalho (1999), we can observe that F_{OC} too is equivalent to F_{GG}.

Now, let us concentrate on F_{DP}. An optimal $L(F_{DP})$ solution of Example 1 has value 3/2 and is shown in Figure 3. The arcs in bold lines are associated with the selected variables (whose values are reported on the arcs). This example is useful for the second relation that we prove.

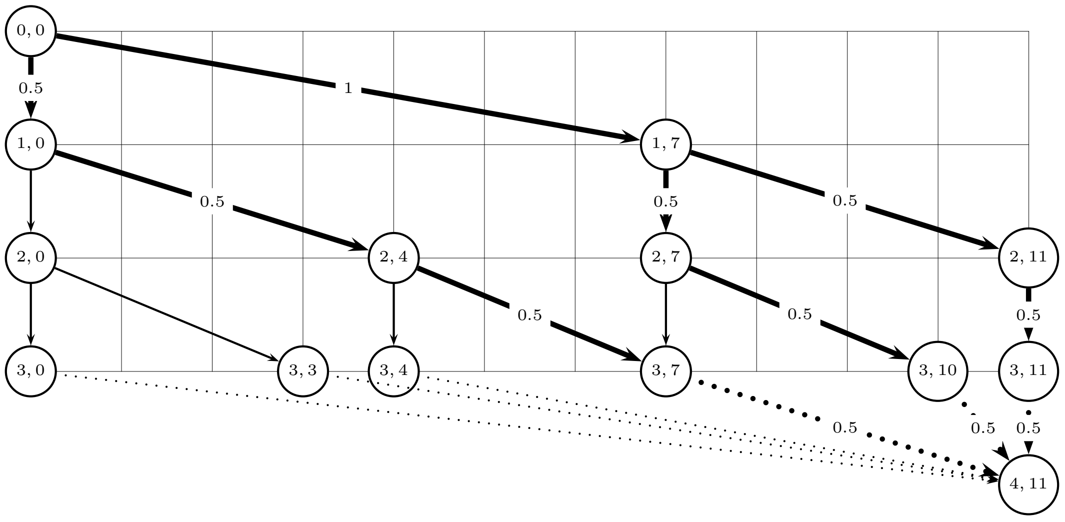


Figure 3: An $L(F_{DP})$ solution of Example 1 (selected arcs in bold, variable values on the corresponding arcs)

Proposition 2 *F_{DP} dominates F_{AF} (and hence F_{OC}).*

Proof. Given in Section Appendix D. □

The relations among the pseudo-polynomial formulations are graphically depicted in the right part of Figure 4. The discussed equivalence among F_{AF}, F_{OC}, and F_{GG} is depicted by the use of a dotted line. The other dotted line depicts our next result.

Proposition 3 *F_{DP} is equivalent to F_{PR}.*

Proof. Given in Section Appendix E. □

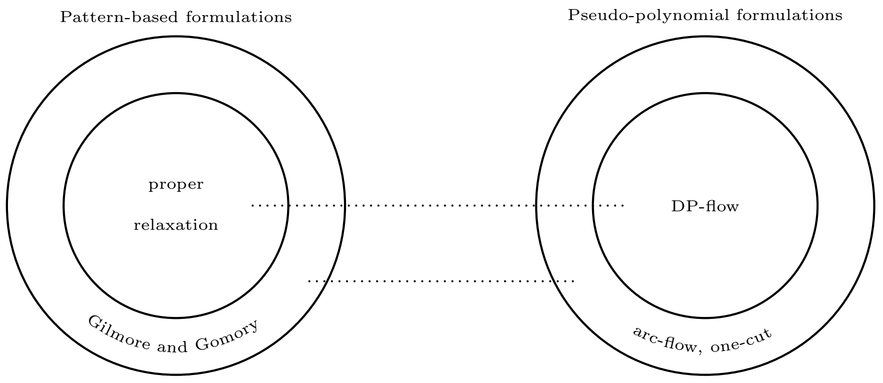


Figure 4: Graphical representation of relations among CSP formulations.

We conclude this section with some remarks. Figure 4 does not depict the "descriptive" CSP formulation (having integer variables for the assignments items-bins and binary variables for the use of the bins), which is largely dominated by all other formulations. As previously noted, pseudo-polynomial formulations were proposed together with some reduction criteria. Dyckhoff (1981) reduced the size of set $\mathcal{S}$ in (3) by forcing $x_j \le d_j$, thus focusing on the set of *normal patterns*:

$$\mathcal{N} = \left\{\bar{w} = \sum\nolimits_{j=1}^{m} w_j x_j,\ \bar{w} \le c,\ x_j \in \mathbb{N}, x_j \le d_j \text{ for } j = 1, \dots, m\right\}. \tag{15}$$

This improves F_{OC} but does not make it equivalent to F_{PR} (consider that the $L(F_{OC})$ solution of Example 1 already accomplishes with (15) but is worse than the $L(F_{DP})$ solution). Valério de Carvalho (1999) proposed a set of reduction techniques for F_{AF}. In particular, he built the arcs by using a DP algorithm that considers item types according to decreasing width, thus reducing symmetries. Also in this case, the continuous relaxation of the resulting formulation is not as strong as that of F_{DP} (once again, the $L(F_{AF})$ solution of Example 1 already accomplishes with the proposed reduction but is worse than the $L(F_{DP})$ solution). Formulation F_{DP} provides a very strong lower bound, but requires too many variables. The tests in Delorme et al. (2016) showed indeed that the computational performance of F_{DP} is much weaker than that of F_{AF} and F_{OC}.

Other related formulations having a good performance have been recently presented. Brandão and Pedroso (2016) developed, among other improvements, a lifting procedure that is based on the fact that packing an item in a position might lead to an unused space in the bin. For each arc, their procedure estimates the minimum unused space by solving a subset sum (as in, e.g., Boschetti and Montaletti 2010 and Dell'Amico et al. 2012) and then uses this value to extend the head of the arc. They are forced to use a multigraph, as arcs having the same widths may correspond to different items, but obtain a good speed-up in the solution time. Clautiaux et al. (2017) proposed a method that iteratively aggregates and disaggregates nodes of an arc-flow model to produce both upper and lower bounds, eventually terminating with a proof of optimality. Their method was computationally evaluated on some cutting stock and vehicle routing instances. Côté and Iori (2016) considered the normal patterns in (15), but conveniently decreased their number by means of a *meet-in-the-middle* procedure. Instead of performing a classical DP, they solved a two-way DP which created patterns starting both from the left and from the right. They then built a reduced arc-flow formulation that required less variables, less constraints, and a quicker solution time. These cited formulations lie in the (small) space between F_{AF} and F_{DP}, the same holds for the formulation that we present in the next section.

Côté and Iori (2016) also proposed to modify F_{AF} by removing unit-width loss arcs $(d, d+1)$ and creating longer loss arcs that connect each vertex in $\mathcal{N}$ to its consecutive vertex in $\mathcal{N}$, and the last vertex in $\mathcal{N}$ to c if $c \notin \mathcal{N}$ (as shown in Figure 5). This implies reducing constraints (8) to $e \in \mathcal{N} \cup \{c\}$ and adopting a multigraph structure (because there can be item and loss arcs having the same head and tail). We embed this straightforward improvement in all next formulations.

4 Reflect, an improved arc-flow formulation

In this section, we propose a new formulation, called *reflect* (F_{RE}), which models a CSP instance by considering only half of the bin capacity, thus resulting in a sharp decrease in the required number of variables and constraints. The two main features of F_{RE} are:

- In terms of vertices, F_{RE} considers only those corresponding to partial bin fillings with size smaller than $c/2$, plus an additional vertex, called R, corresponding to $c/2$;
- In terms of arcs, F_{RE} considers the same ones of F_{AF}, but: (i) "reflects" each item arc (d, e) having $d < c/2$ and $e > c/2$ into an arc $(d, c - e)$; (ii) removes all item and loss arcs (d, e) having $d \geq c/2$; and (iii) creates a last loss arc by connecting the right most vertex before R with R.

Intuitively, a path in F_{AF} becomes in F_{RE} *a pair of colliding paths*, i.e., two paths both starting in 0 and ending in the same vertex, but only one of the two passing through a reflection. For Example 1, the arcs required by F_{AF} and F_{RE} are shown in Figure 5. F_{AF} contains 6 items arcs and 5 loss arcs. To build F_{RE}, (i) we reflect item arcs (0,7) in (0,4) and (4,7) in (4,4); (ii) we remove item arcs (7,10) and (7,11) and loss arcs (7,10) and (10,11); and (iii) we replace loss arc (4,7) with (4,R) and insert (R, R), thus resulting in 4 item arcs and 4 loss arcs.

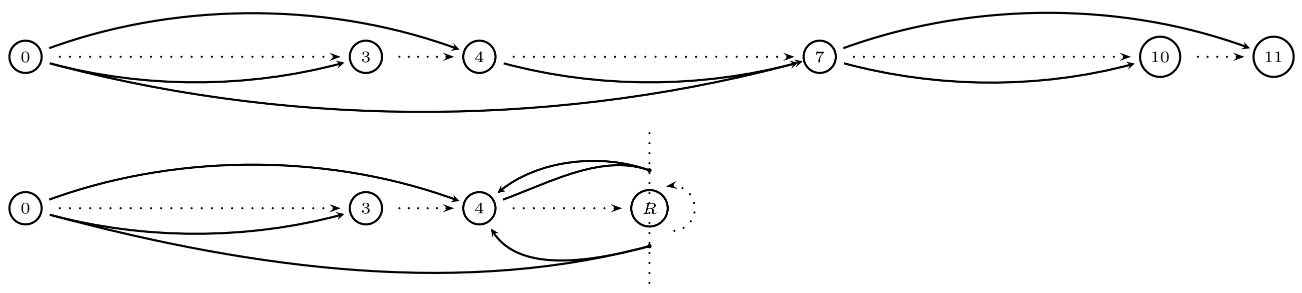


Figure 5: Multigraphs required by arc-flow (above) and reflect (below) for Example 1 (item arcs are depicted in solid lines, loss arcs in dotted lines)

The formulation is built on a multigraph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$. The set $\mathcal{V}$ of vertices comprises all partial bin fillings between 0 and $c/2$ that correspond to arc tails and heads. The set of arcs $\mathcal{A}$ is partitioned into $\mathcal{A}_s$ and $\mathcal{A}_r$, where $\mathcal{A}_s$ denotes the set of *standard arcs*, i.e., all those item and loss arcs that proceed from left to right as in F_{AF}, and $\mathcal{A}_r$ the set of *reflected arcs*, i.e., those item arcs (d, e) from F_{AF} that have been reflected into item arcs $(d, c-e)$. Each arc in $\mathcal{A}_s$ is defined by the triplet (d, e, s), whereas each arc in $\mathcal{A}_r$ by the triplet (d, e, r) (note that there can be standard and reflected arcs having the same head and tail). We include in $\mathcal{A}_r$ an arc $(c/2, c/2, r)$ to model pairs of paths that collide in $c/2$. We use (d, e, κ) to denote a generic arc belonging to either $\mathcal{A}_s$ or $\mathcal{A}_r$, and $\mathcal{A}_j = \{(d, d+w_j, s) \in \mathcal{A}_s\} \cup \{(d, c-d-w_j, r) \in \mathcal{A}_r\}$ to define the set of item arcs associated with item type j. Let also $\delta_s^-(e) \subseteq \mathcal{A}_s$ (respectively, $\delta_r^-(e) \subseteq \mathcal{A}_r$) denote the set of standard (respectively, reflected) arcs entering e. By associating an integer variable $\xi_{de\kappa}$ with each $(d, e, \kappa) \in \mathcal{A}$, we can model the CSP as

$$(F_{RE}) \quad \min \quad z = \sum_{(d,e,r)\in\mathcal{A}_r} \xi_{der} \tag{16}$$

$$\text{s.t.} \quad \sum_{(d,e,s)\in\delta_s^-(e)} \xi_{des} = \sum_{(d,e,r)\in\delta_r^-(e)} \xi_{der} + \sum_{(e,f,\kappa)\in\delta^+(e)} \xi_{ef\kappa} \quad e \in \mathcal{V} \setminus \{0\} \tag{17}$$

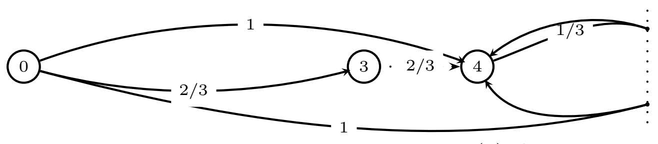
$$\sum_{(0,e,\kappa)\in\delta^+(0)} \xi_{0e\kappa} = 2 \sum_{(d,e,r)\in\mathcal{A}_r} \xi_{der} \tag{18}$$

$$\sum_{(d,e,\kappa)\in\mathcal{A}_j} \xi_{de\kappa} \geq d_j \quad j = 1, \ldots, m \tag{19}$$

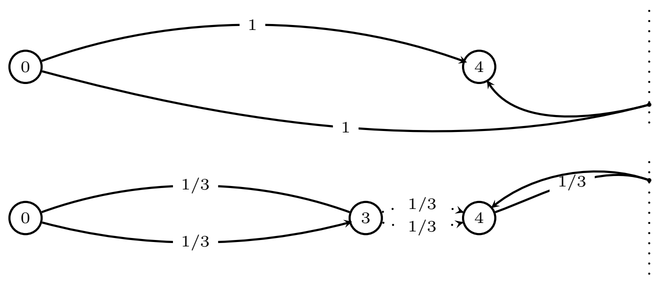
$$\xi_{de\kappa} \in \mathbb{N} \quad (d, e, \kappa) \in \mathcal{A}. \tag{20}$$

Objective function (16) minimizes the number of reflected arcs, which is equivalent to the number of bins. Constraints (17) ensure that the flow on standard arcs entering a node e is equal to the flow (on both standard and reflected arcs) emanating from e plus the flow on reflected arcs entering e. Constraints (18) impose boundary conditions by enforcing the flow emanating from 0 to be twice the number of bins, and constraints (19) ensure that demands are fulfilled.

An optimal $L(F_{RE})$ solution of Example 1 having value 4/3 is given in Figure 6-(a), and can be decomposed into the pairs of colliding paths shown in Figure 6-(b). The first pair is made by paths $(0, 4, s)$ and $(0, 4, r)$, which collide in 4 and corresponds to a bin containing an item of width 4 and another of width 7. The second pair is made by arcs $(0, 3, s)$, $(3, 4, s)$, and $(4, 4, r)$. Note that arcs $(0, 3, s)$ and $(3, 4, s)$ are both depicted twice to emphasize that the flow on them is split to form the two colliding paths, the first being $\{(0,3,s),(3,4,s),(4,4,r)\}$ and the second $\{(0,3,s),(3,4,s)\}$, both with flow 1/3. Note also that, if no reflected arc enters e, then e is just a partial filling of one or more bins (e.g., vertex 3 in the example), if instead some reflected arcs enter e, then e is a vertex of collision for one or more bins (e.g., vertex 4).



(a) An optimal $L(F_{RE})$ solution



(b) Decomposition of the solution above

Figure 6: An optimal $L(F_{RE})$ solution of Example 1 and its decomposition into pairs of colliding paths (selected item arcs depicted in solid lines and selected loss arcs in dotted lines, variable values on the arcs)

The following result proves the correctness of F_{RE}.

Proposition 4 *F_{RE} models the CSP.*

Proof. Given in Section Appendix F. □

For the sake of completeness, we provide in Algorithm 7, Section Appendix G, the procedure that we use to construct the multigraph required by F_{RE}. In the same section, we also provide in Algorithm 8 the procedure that we use to decompose a F_{RE} (or $L(F_{RE})$) solution into pairs of colliding paths. One can notice that Algorithm 7 does not create any reflected arc (d,e,r) having $d > e$ (step 15 of the algorithm). This reduction criterion is motivated by the following result.

Proposition 5 *Any feasible CSP pattern can be represented in F_{RE} by a pair of colliding paths whose reflected arc (d,e,r) has $d \leq e$.*

Proof. Given in Section Appendix F. □

4.1 Adapting reflect to solve large size instances: Reflect+

Even if the number of arcs used by F_{RE} is considerably reduced with respect to those required by F_{AF}, some instances with huge capacity and many small items may still generate models that contain millions of variables and are thus too difficult to tackle. To overcome this issue, we propose some lower and upper bounding techniques and embed them into a new algorithm, called *reflect+*.

Column generation. We first solve $L(F_{PR})$, the linear relaxation of (2), by means of a standard column generation technique. The reduced master problem is initialized with the identity matrix and solved as a linear program to obtain dual variable values $\bar{\pi}_j$ for each item type j. Columns with negative reduced costs are found and added to the reduced master on the fly, by solving a knapsack subproblem, until a proof of optimality is reached. For the subproblem, we make use of `combo` by Martello et al. (1999), which solves the binary knapsack. We first use `combo` as a heuristic by feeding it with m items j of profit $\bar{\pi}_j$ and weight w_j (just one item per item type). If this attempt fails in finding a negative reduced cost column, we use `combo` as an exact by feeding it with the entire set of items, but invoking a binary expansion (see, e.g., Vanderbeck and Wolsey 2010). Each item type j having demand d_j is represented by $\lfloor \log d_j \rfloor + 1$ items, the first items $k = 0, 1, \ldots, \lfloor \log d_j \rfloor - 1$ have profit $2^k \pi_j$ and weight $2^k w_j$, and the last item $l = \lfloor \log d_j \rfloor$ has profit $(d_j - (2^l - 1))\pi_j$ and weight $(d_j - (2^l - 1))w_j$. The first heuristic has the purpose of avoiding patterns with many small items that can appear in early column generation iterations and slightly deteriorate our successive upper bounding procedures.

Let $P_{LP} \subseteq P$ denote the set of columns that have been generated to reach linear optimality. A classical way to obtain an upper bound from this information is to solve to optimal integrality the restricted master problem with just the set P_{LP} of columns. This procedure, usually known as *restricted master heuristic*, is easy to implement but might produce low quality solutions (see, e.g., Sadykov et al. 2016). Here, we propose a simple yet effective improvement that consists in solving F_{RE} on a multigraph that contains only the arcs produced by P_{LP}. In detail, we start with the empty F_{RE} multigraph, consider all items contained in a column p by non-increasing weight, and generate a single arc for each item in that order (using the algorithm sketched in the proof of Proposition 5). We repeat the process for all $p \in P_{LP}$ and then solve F_{RE} on the reduced instance. Our method is motivated by the following

Remark 1 *Let $z(F_{PR}(P_{LP}))$ be the optimal solution value of the restricted F_{PR} that contains only the columns in P_{LP}, and $z(F_{RE}(P_{LP}))$ be the optimal solution value of the restricted F_{RE} that contains only the arcs produced by the columns in P_{LP}, then $z(F_{RE}(P_{LP})) \leq z(F_{PR}(P_{LP}))$.*

The remark follows from the fact that all patterns $p \in P_{LP}$ can be produced by F_{RE}, but F_{RE} can also produce patterns that do not belong to P_{LP}. Consider for example the bottom part of Figure 5. Those arcs may be obtained through the mapping of patterns (1,0,1) and (0,1,1), but it is possible for F_{RE} to also produce the proper pattern (1,1,0) through arcs $\{(0,4,r),(0,4,s)\}$, and the non-proper patterns (0,2,1) through arcs $\{(0,4,s),(4,4,r),(0,4,s)\}$, (0,1,2) through arcs $\{(0,4,s),(4,4,r),(0,3,s), (3,4,s)\}$, and (0,0,3) through arcs $\{(0,3,s),(3,4,s),(4,4,r),(0,3,s),(3,4,s)\}$, thus providing a large number of possible heuristic solutions.

In our implementation, we first compute $P_{base} \subseteq P_{LP}$ as the subset of columns whose associated variables take positive value in the linear solution. We then compute $z(F_{RE}(P_{base}))$, which is usually quick, and then $z(F_{RE}(P_{LP}))$ if needed.

Node deactivation and dual cuts. We solve $L(F_{RE})$ with the complete set of arcs, and then use its linear solution $\bar{\xi}$ to build the set of *non-active* vertices $\mathcal{V}_n = \{d \in \mathcal{V} : \nexists\ \bar{\xi}_{(d,e,\kappa)} > \epsilon\}$. Then, we solve F_{RE} with the additional constraints

$$\xi_{de\kappa} = 0 \quad d \in \mathcal{V}_n, (d, e, \kappa) \in \mathcal{A}. \tag{21}$$

Constraints (21) make the solution of the model much faster but might remove too many feasible solutions. We experimentally noticed that better solutions could be found by allowing some large items to be split into smaller items. To this aim, we create a set T of possible transformations (i, j, k), in which $i, j, k = 1, \ldots, m$, $i < j \leq k$, and $w_i = w_j + w_k$. We create a family of integer variables t_{ijk}, for $(i, j, k) \in T$, each counting how many times an item i is transformed into items j and k. We then replace (19) with

$$\sum_{(d,e,\kappa)\in\mathcal{A}_j} \xi_{de\kappa} + \sum_{(i,j,k)\in T} t_{ijk} + \sum_{(i,k,j)\in T} t_{ikj} - \sum_{(j,k,i)\in T} t_{jki} \geq d_j \quad j = 1, \ldots, m \tag{22}$$

$$t_{ijk} \in \mathbb{N} \qquad (i, j, k) \in T \tag{23}$$

and solve model (16)–(18), (20)–(23). This procedure is reminiscent of the dual cuts by Valério de Carvalho (2005).

Arc deactivation. Our third heuristic attempts to find a solution having exactly value $L = \lceil L(F_{RE}) \rceil$. To this aim, we gather in a set $\mathcal{A}_z$ all arcs whose reduced cost is greater than $L - L(F_{RE}) + \epsilon$, and restrict the F_{RE} model by setting

$$\xi_{de\kappa} = 0 \quad (d, e, \kappa) \in \mathcal{A}_z. \tag{24}$$

Indeed, selecting one or more of arcs in $\mathcal{A}_z$ would imply a solution value greater than L. We then solve model (16)–(20), (24). If no solution of value L is found, we increase L by one, update $\mathcal{A}_z$, and iterate. Note that, if the MIRUP conjecture holds, then the process is iterated at most once.

The resulting reflect+ algorithm makes use of these techniques in the order in which we presented them. An informal pseudocode is given in Algorithm 1. Although not explicitly stated, the algorithm clearly stops as soon as upper and lower bound values are equal. Each time a model is solved as an MILP, it is allowed only a restricted execution time, as discussed in Section 7 below.

Algorithm 1 `reflect+`

```
1: solve $L(F_{PR})$ through column generation and obtain $P_{LP}$, $P_{base}$, and $L = \lceil L(F_{PR}) \rceil$
2: solve $F_{RE}(P_{base})$ and obtain $U_1$
3: solve $F_{RE}(P_{LP})$ and obtain $U_2$
4: solve $L(F_{RE})$ and obtain $\bar{\xi}$, $\mathcal{V}_n$, and $\mathcal{A}_z$
5: solve $F_{RE}$ + (21) − (23) and obtain $U_3$
6: $U \leftarrow \min(U_1, U_2, U_3)$
7: while $U > L$ do
8:     solve $F_{RE}$ + (24) and obtain $U_4$
9:     $U = \min(U, U_4)$
10:    update $\mathcal{A}_z$ and set $L \leftarrow L + 1$
11: end while.
```

5 The Variable Sized BPP

In this section, we detail the modifications that we developed to solve the *variable sized bin packing problem* (VSBPP). In the VSBPP, we are given a set T of bin types, each having capacity c_t, cost p_t, and a number b_t of available bins, for $t \in T$. The aim is to pack all items into a set of bins of minimum cost. The solution of the VSBPP (which is also known in the literature as multiple length CSP) has been attempted through several methods, including arc-flow (Valério de Carvalho 2002), branch-and-cut-and-price (Belov and Scheithauer 2002, Alves and Valério de Carvalho 2008), reformulations by discretization (Correia et al. 2008), and methaheuristics (Hemmelmayr et al. 2012), among others.

Valério de Carvalho (2002) described how to adapt F_{AF} to the VSBPP by using a vertex set from 0 to c, with $c = \max_t\{c_t\}$, and an additional family of integer variables ω_t, giving the number of bins of type t selected in the solution, for $t \in T$. The total flow emanating from 0 is set to $\sum_{t \in T} \omega_t$, whereas the difference between incoming and emanating flow at each bin capacity c_t is not forced anymore to be 0, as in the other vertices, but is equal to ω_t. The resulting flow can be decomposed into a set of paths ending at a c_t position and thus being associated with a specific bin type.

Such adaptation is not straightforward for reflect, where we need to enforce ω_t to be equal to the number of selected pairs of colliding paths of total length c_t. We can meet this requirement, while still using half of the nodes, by associating ω_t with the reflected arcs and using the fact that just one such arc is used for each colliding pair. In detail, we create a multigraph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ that is similar to the one adopted for F_{RE} but contains a few relevant modifications. Apart from the vertices obtained by item widths combinations and reflections, $\mathcal{V}$ now also includes the set $\{c_t/2 : t \in T\}$. The arc set $\mathcal{A}$ is still partitioned in $\mathcal{A}_s$ and $\mathcal{A}_r$, where $\mathcal{A}_s$ is created as in F_{RE}, but $\mathcal{A}_r$ is enlarged to take into account reflections at all possible bin capacities, namely, $\mathcal{A}_r = \cup_{t \in T} \mathcal{A}_r(t)$, where $\mathcal{A}_r(t)$ is the set of arcs that have been reflected at $c_t/2$. We also use the notation $(d, e, r(t))$ to denote an arc belonging to $\mathcal{A}_r(t)$. The algorithm that we implemented to build $\mathcal{G}$ is given in Section Appendix H. We explain here the intuition behind our formulation by means of the following example.

Example 2 *A VSBPP instance with two bin types and three item types, having $c = (11, 6)$, $p = (11, 6)$, $b = (2, 2)$, $w = (7, 4, 3)$, and $d = (1, 1, 1)$.*

The arcs required by reflect to model Example 2 are shown in Figure 7. The vertices used for reflection are 3 (for bins of size 6) and $R = c/2$ (for bins of size c_2 =11). Item 1 is represented by arc $(0, 4, r(1))$, item 2 by arcs $(0, 4, s)$ and $(0, 2, r(2))$, and item 3 by arcs $(0, 3, s)$ and $(4, 4, r(1))$. Loss arcs are created between pairs of consecutive vertices. The arc set is completed by $(3, 3, r(2))$ and $(R, R, r(1))$ to model pairs of paths that collide in 3 or in R. An optimal solution has cost 17 and consists of two pairs of paths: $(0, 4, s)$ and $(0, 4, r(1))$, which correspond to a bin of type 1 containing items 1 and 2; and $(0, 3, s)$, $(0, 2, s)$, $(2, 3, s)$, and $(3, 3, r(2))$, which corresponds to a bin of type 2 containing item 3.

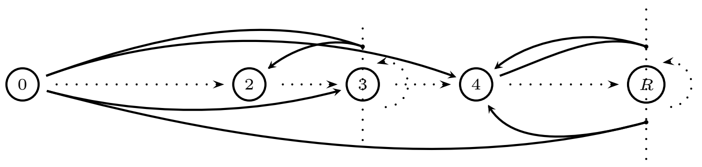


Figure 7: Multigraph required by reflect for Example 2 (item arcs depicted in solid lines, loss arcs in dotted)

By associating once more an integer variable $\xi_{de\kappa}$ to each arc, we can model the VSBPP as

$$(F_{RE}^{VS}) \quad \min \quad z = \sum_{t \in T} p_t \omega_t \tag{25}$$

$$\text{s.t.} \quad \sum_{(d,e,s) \in \delta_s^-(e)} \xi_{des} = \sum_{(d,e,r) \in \delta_r^-(e)} \xi_{der} + \sum_{(e,f,\kappa) \in \delta^+(e)} \xi_{ef\kappa} \quad e \in \mathcal{V} \setminus \{0\} \tag{26}$$

$$\sum_{(0,e,\kappa) \in \delta^+(0)} \xi_{0e\kappa} + \sum_{(d,0,r) \in \delta^-(0)} \xi_{d0r} = 2 \sum_{t \in T} \omega_t \tag{27}$$

$$\sum_{(d,e,r(t)) \in \mathcal{A}_r(t)} \xi_{der(t)} = \omega_t \quad t \in T \tag{28}$$

$$\sum_{(d,e,\kappa) \in \mathcal{A}_j} \xi_{de\kappa} \geq d_j \quad j = 1, \ldots, m \tag{29}$$

$$\xi_{de\kappa} \in \mathbb{N} \quad (d, e, \kappa) \in \mathcal{A} \tag{30}$$

$$\omega_t \in \{0, 1, \ldots, b_t\} \quad t \in T. \tag{31}$$

Objective function (25) minimizes the cost of the selected bins, whereas constraints (26) impose flow conservation. Constraints (27) still force the amount of flow emanating from 0 to be twice the number of bins used, but takes into account the fact that it is now possible for a reflected arc to directly enter vertex 0 (in case $\exists\ j$ and t for which $w_j = c_t$). Constraints (28) impose ω_t to be equal to the flow on the reflected arcs in $\mathcal{A}_r(t)$, and constraints (29) ensure that demands are met. Note that variables ω_t are not mandatory because of (28), but their use proved to be computationally useful, especially for instances where p_t values were not proportional to c_t ones.

Algorithm reflect+ is adjusted to cope with the VSBPP as follows. At step 1 (refer to Algorithm 1), a VSBPP column generation procedure is invoked (we use the approach described in Section 3.1 of Alves and Valério de Carvalho 2008, and adopt `combo` by Martello et al. 1999 for the solution of the slavhee problems). At steps 2 and 3, restricted versions of F_{RE}^{VS} are solved instead of F_{RE}. At step 4, $L(F_{RE}^{VS})$ is invoked, whereas step 5 is skipped because of poor computational performance. The main loop is modified by invoking F_{RE}^{VS} at step 8 and increasing L to $(\min \sum_{t\in T} p_t\omega_t : \sum_{t\in T} p_t\omega_t > L, \omega_t \in \{0, 1, \ldots, b_t\}$ for $t \in T)$ at step 10. We call reflect$_+^{VS}$ the resulting algorithm.

6 BPP with item fragmentation

The *bin packing problem with item fragmentation* (BPPIF) is the BPP generalization in which items are allowed to be fractionally packed into different bins. Several BPPIF variants have been studied in the literature, see, e.g., Casazza and Ceselli (2016). In this section, we show how to solve the two main problem variants: (i) *minimize the number of bins* used for the packing while the total number of fragmentations is at most F (bm-BPPIF); and (ii) *minimize the number of fragmentations* while the number of bins is at most B (fm-BPPIF). For both variants, we first propose an extension of the classical arc-flow model which produces good lower bounds, and then show how to adapt F_{RE}^{VS} to derive good quality upper bounds. Without loss of generality, we suppose that $w_j < c$ holds for any j.

In our new arc-flow model, arcs can exceed c and re-enter the bin from 0. As in the previous sections, our approach starts by constructing the multigraph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ that is at the basis of the formulation. In this case, set $\mathcal{V}$ contains vertices between 0 and c. When attempting to create an arc (d, e) having $e > c$, its head e is moved to $e \mod c$, resulting in a *transposed arc*. The arc set is consequently divided in $\mathcal{A} = \mathcal{A}_s \cup \mathcal{A}_t$, where $\mathcal{A}_s$ is the set of standard arcs and $\mathcal{A}_t$ the set of transposed arcs. Once again, for the sake of conciseness we provide the algorithm to build $\mathcal{G}$ in the appendix (Section Appendix I), but provide here a clarifying example.

Example 3 *A BPPIF instance with $m = 2$, $w = (7, 5)$, $d = (2, 1)$, and $c = 11$.*

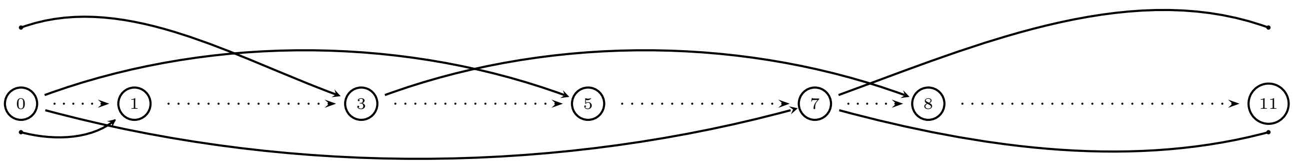


Figure 8: Multigraph required by BPPIF arc-flow for Example 3 (item arcs are depicted in solid lines)

For Example 3, the arcs required by our arc-flow formulations are shown in Figure 8. A first arc (0,7) is created by the first item of width 7. An arc (7,14) is created by the second item of width 7 and is transposed into (7,3). Then, arcs (0,5), (3,8), and the transposed (7,1) are created by the item of width 5. Finally, loss arcs linking together consecutive vertices are added. An optimal solution for the bm-BPPIF having $F = 1$ uses two bins and corresponds to arcs (0,7), (7,3), (3,8), and (8,11). The same solution is optimal also for the fm-BPPIF having $B = 2$.

Let $\mathcal{A}_j = \{(d, d + w_i) \in \mathcal{A}_s\} \cup \{(d, (d + w_i) \mod c) \in \mathcal{A}_t\}$. The bm-BPPIF can be modeled as

$$(F_{AF}^{bm}) \quad \min \quad z + \sum_{(d,e)\in\mathcal{A}_t} x_{de} \tag{32}$$

$$\text{s.t.} \quad \sum_{(e,f)\in\delta^+(e)} x_{ef} - \sum_{(d,e)\in\delta^-(e)} x_{de} = \begin{cases} z & \text{if } e = 0 \\ -z & \text{if } e = c \\ 0 & \text{for } e \in \mathcal{V}, 0 < e < c \end{cases} \tag{33}$$

$$\sum_{(d,e)\in\mathcal{A}_j} x_{de} \geq d_j \qquad j = 1, \dots, m \tag{34}$$

$$\sum_{(d,e)\in\mathcal{A}_t} x_{de} \leq F \tag{35}$$

$$x_{de} \in \mathbb{N} \qquad (d, e) \in \mathcal{A}. \tag{36}$$

Objective function (32) minimizes the number of bins, which is equal to the flow emanating from 0 plus the flow on the transposed arcs. Constraints (35) ensure that the number of fragmentations is not greater than F. The adaptation of this model to the fm-BPPIF variant is

$$(F_{AF}^{fm}) \quad \left\{ \min \sum\nolimits_{(d,e)\in\mathcal{A}_t} x_{de} : \ (33), (34), (36), \text{ and } z + \sum\nolimits_{(d,e)\in\mathcal{A}_t} x_{de} \leq B \right\} \tag{37}$$

Model (37) minimizes the number of fragmentations (instead of limiting it to F, as in (35)) and imposes a feasible flow containing all items. The last part of (37) forces the use of at most B bins.

As it will be shown by the experimental evaluation in Section 7, these formulations are useful to obtain good quality lower bounds but quite weak in quickly finding feasible solutions. We could not find a direct formulation of the problem based on reflect, but, with the aim of obtaining good quality upper bounds, we implemented an approach based on tailored VSBPP instances solved by slightly modified F_{RE}^{VS} formulations. Let us describe it first for the bm-BPPIF. Let the bin types in set T be numbered as $t = 1, 2, \dots, |T|$. For each bin type t, we define an arbitrarily large availability b_t, a capacity $c_t = tc$, and a cost $p_t = t$. Under this construction, using a bin of type t in the VSBPP corresponds to using t bins of capacity c in the BPPIF. This derives from the fact that a fractional packing in t bins can always be obtained by using no more than $t - 1$ fragmentations (see, e.g., Casazza and Ceselli 2016). Thus, an optimal bm-BPPIF solution can be obtained by solving a modified F_{RE}^{VS} that includes the additional constraint $\sum_{t=1}^{|T|}(t-1)\omega_t \leq F$.

When $|T|$ is sufficiently large, this approach leads to an optimal solution. However, this may result in a long computing time because of the consequent large capacities involved. Our approach, called reflect^{IF}, attempts instead different increasing values of $|T|$ in the set $\{1, 2, 3\}$. For each value, it invokes the modified F_{RE}^{VS} with a limited time and checks whether the solution found (if any) is proven optimal. If not, it continues to the next T value. If the three attempts are concluded without a proof of optimality, then reflect^{IF} invokes F_{AF}^{bm}. The adaptation of this approach to the fm-BPPIF is easy, as it only requires to modify F_{RE}^{VS} by setting the objective function to $\sum_{t=1}^{|T|}(t-1)\omega_t$ and imposing the additional constraint $\sum_{t=1}^{|T|} t\omega_t \leq B$, and then invoking F_{AF}^{fm} in the last step. We opted not to modify the advanced bounding procedures for the large-size instances (Section 4.1) to cope with the BPPIF, because, as shown in Section 6 below, the approach that we just described could optimally solve all benchmarks.

7 Computational results

In this section, we computationally evaluate the proposed techniques and compare them with the existing literature. All our experiments have been executed on an Intel Xeon 3.10 GHz with 8 GB RAM (having CPU passmark indicator = 6594 in `www.passmark.com`), using Gurobi 6.5 as MILP solver. All tests were performed with a single core and by setting the number of threads to one in the MILP solver configuration. Due to the large number of instances attempted, in the following tables we mostly provide aggregate information for each algorithm and each group of instances, including the two following main indicators:

- #opt = total number of proven optimal solutions among the instances in the line; and
- sec = average of the CPU seconds elapsed across all instances in the line.

The best values of #opt in each line are highlighted in bold. When computing sec, we consider the time limit value for all instances that were not solved to proven optimality. Lines marked total/avg report total #opt and average sec values for groups of two or more lines.

7.1 Results on BPP and CSP

We tested our algorithms on the most-well known and challenging BPP and CSP benchmark sets:

(1) Classical: A set of 1615 instances proposed in various articles in the last decades and having variegate characteristics (a complete description is given in Delorme et al. 2016);

(2) GI: 4 sets of 60 instances proposed by Gschwind and Irnich (2016) and involving bin capacities up to 1 500 000;

(3) AI/ANI: 2 sets of 100 challenging instances proposed by Delorme et al. (2016).

Instances and algorithms of this section are available at the BPPLIB by Delorme et al. (2017a). We first evaluate the entity of the improvement obtained by F_{RE} on the classical F_{AF}. Table 1 provides the results obtained by running the two formulations with a time limit of 3600 seconds. The first two columns identify the benchmark set and the corresponding number of instances. Apart from #opt and sec, the table gives the average number of variables (nb. var.) and constraints (nb. cons.) in the models.

Table 1: Comparison between the classical arc-flow and the new reflect formulations on BPP/CSP benchmarks

set of instances	#inst.	arc-flow (F_{AF})				reflect (F_{RE})			
		#opt	sec	nb. var.	nb. cons.	#opt	sec	nb. var.	nb. cons.
Waescher	17	9	1780.5	174 722	9256	**17**	555.5	52 006	4257
Hard28	28	**28**	10.9	36 816	1134	**28**	19.0	10 932	635
Falkenauer U	80	**80**	0.1	3023	205	**80**	0.0	765	131
Falkenauer T	80	**80**	1.3	16 246	735	**80**	0.3	2490	332
Schwerin 1	100	**100**	0.9	11 636	733	**100**	0.3	3408	287
Schwerin 2	100	**100**	0.7	12 442	739	**100**	0.2	3664	292
Scholl 1	720	**720**	0.1	1735	166	**720**	0.0	510	113
Scholl 2	480	**480**	84.3	39 307	938	**480**	9.6	13 187	453
Scholl 3	10	**10**	324.2	1 529 969	49 268	**10**	8.6	29 938	10 923
total/avg (1)	1615	1607	46.2	26 853	913	**1615**	9.1	5668	367
GI AA	60	20	2699.3	4 878 777	205 208	**60**	179.8	82 069	23 436
GI AB	60	0	3600.0	19 852 031	441 111	0	3600.0	4 493 237	191 111
GI BA	60	20	2727.3	10 113 134	510 332	**60**	380.1	99 396	39 627
GI BB	60	0	3600.0	52 020 717	1 281 154	0	3600.0	11 271 290	531 165
total/avg (2)	240	40	3156.7	21 716 164	614 405	**120**	1940.0	3 986 498	201 288
AI 200	50	**50**	233.7	121 251	2249	**50**	45.5	42 803	1140
AI 400	50	19	2461.3	940 036	7686	**21**	2297.4	335 723	3768
ANI 200	50	35	1397.7	119 496	2245	**50**	67.2	42 021	1136
ANI 400	50	3	3474.4	935 117	7683	**10**	3083.6	334 149	3765
total/avg (3)	200	107	1891.8	528 975	4966	**131**	1373.4	188 674	2452
total/avg (1)+(2)+(3)	2055	1754	327.6	2 608 779	72 956	**1866**	190.3	488 393	24 035

It can be noted that F_{RE} outperforms F_{AF} both in terms of #opt and sec. This can be explained by the drastic variable and constraint reductions obtained by reflect: An average 81.2% of variable reduction (ranging from 64.5% for AI 400 up to 98% for Scholl 3); and an average 64.1% of constraint reduction (ranging from 31.9% for Scholl 1 up to 92.2% for GI BA). However, even with these considerable reductions, reflect cannot handle the millions of variables and the hundreds of thousands constraints required to model the GI AB and GI BB instances.

We now analyze the performance of reflect+. We first focus on the difference between the classical restricted master heuristics (that solve F_{PR} with less columns) adopted many times in the literature, and the first two new heuristics used in reflect+ (that solve F_{RE} with less arcs). Table 2 provides the results obtained by running F_{PR} and F_{RE} considering only columns/arcs from P_{base} or P_{LP} (see Section 4.1), with a time limit of 60 seconds. Apart from #opt and sec, the table provides the number of times an execution reached the time limit (t.l.), and the average absolute gap from the best known lower bound (a.g.=$U - L$).

Table 2: Comparison between classical restricted master heuristics ($F_{PR}(P_{base})$ and $F_{PR}(P_{LP})$) and new reflect based heuristics ($F_{RE}(P_{base})$ and $F_{RE}(P_{LP})$) on BPP/CSP benchmarks

set of instances	#inst.	$F_{PR}(P_{base})$				$F_{PR}(P_{LP})$				$F_{RE}(P_{base})$				$F_{RE}(P_{LP})$			
		#opt	sec	#t.l.	a.g.	#opt	sec	#t.l.	a.g.	#opt	sec	#t.l.	a.g.	#opt	sec	#t.l.	a.g.
Waescher	17	0	0.3	0	2.4	5	16.1	2	0.8	7	10.7	2	0.6	**9**	23.7	6	0.5
Hard28	28	0	0.2	0	2.5	5	1.0	0	0.8	0	0.0	0	1.0	**15**	2.7	0	0.5
Falkenauer U	80	12	0.0	0	1.1	67	0.1	0	0.2	79	0.0	0	0.0	**80**	0.0	0	0.0
Falkenauer T	80	10	0.6	0	2.1	17	4.5	1	0.8	10	0.1	0	0.9	**80**	0.6	0	0.0
Schwerin 1	100	0	0.1	0	2.3	16	21.3	8	1.0	**100**	0.0	0	0.0	**100**	0.2	0	0.0
Schwerin 2	100	0	0.2	0	2.2	44	17.1	16	0.6	**100**	0.0	0	0.0	**100**	0.1	0	0.0
Scholl1	720	287	0.0	0	0.8	529	0.2	1	0.3	717	0.0	0	0.0	**720**	0.0	0	0.0
Scholl2	480	9	11.5	67	3.4	111	36.4	266	2.0	474	0.9	3	0.0	**475**	1.6	5	0.0
Scholl3	10	0	5.0	0	8.8	0	60.1	10	5.1	**10**	0.3	0	0.0	**10**	1.9	0	0.0
total/avg (1)	1615	318	3.5	67	1.9	794	14.1	304	0.9	1497	0.4	5	0.1	**1589**	0.8	11	0.0
GI AA	60	4	19.3	19	2.0	32	22.1	20	0.9	**60**	0.5	0	0.0	**60**	2.5	0	0.0
GI AB	60	3	32.3	28	2.5	24	39.2	36	1.6	**59**	1.6	0	0.0	56	14.3	4	0.3
GI BA	60	6	17.7	16	1.9	33	22.4	20	0.8	59	0.6	0	0.0	**60**	2.0	0	0.0
GI BB	60	2	32.6	29	2.6	23	38.1	35	1.7	**59**	3.5	1	0.0	50	16.3	10	0.7
total/avg (2)	240	15	25.4	92	2.3	112	30.5	111	1.2	**237**	1.5	1	0.0	226	8.8	14	0.2
AI 200	50	1	0.1	0	2.5	1	1.2	0	1.0	1	0.0	0	1.0	**13**	2.2	0	0.7
AI 400	50	0	9.5	2	5.7	0	49.1	36	1.9	0	0.6	0	1.0	0	19.6	0	1.0
ANI 200	50	0	0.0	0	2.2	0	0.9	0	1.0	0	0.0	0	1.0	0	1.5	0	1.0
ANI 400	50	0	8.9	1	5.5	0	53.4	40	1.8	0	0.6	0	1.0	0	23.0	3	1.1
total/avg (3)	200	1	4.6	3	3.9	1	26.1	76	1.4	1	0.3	0	1.0	**13**	11.6	3	1.0
total/avg (1)+(2)+(3)	2055	333	6.4	359	2.0	906	16.2	615	1.0	1734	0.5	206	0.1	**1815**	1.9	225	0.0

Table 2 shows that, when used on a restricted set of patterns, F_{RE} largely outperforms F_{PR}. Sometimes, both $F_{PR}(P_{base})$ and $F_{PR}(P_{LP})$ are fast but tend to terminate with sub-optimal solutions (e.g., on Waescher), while some other times they do not finish within the time limit (e.g., on Scholl 2). Both $F_{RE}(P_{base})$ and $F_{RE}(P_{LP})$ obtain good results as they solve to optimality most of the instances within a small time, achieving a small average gap on the unsolved instances. There is not a clear dominance between $F_{RE}(P_{base})$ and $F_{RE}(P_{LP})$: $F_{RE}(P_{base})$ is faster, especially for instances with very large capacity (e.g., GI AB), but $F_{RE}(P_{LP})$ terminates more often with a proven optimal solution (e.g., Falkenauer T). Note that no algorithm is capable of providing good results for the difficult AI/ANI instances, for which more advanced techniques are needed.

Table 3 compares reflect and reflect+ with the best algorithms available in the BPP/CSP literature. On the basis of the results in Delorme et al. (2016), we selected the two approaches that largely dominated other 10 exact methods, namely, the branch-and-cut-and-price by Belov and Scheithauer (2006), simply called *Belov* hereafter, and the *VPSolver* by Brandão and Pedroso (2016). We ran the codes of the two methods on our machine. As required by the codes, Cplex (version 12.6) was used for Belov and Gurobi (6.5) for VPSolver. In addition to #opt and sec, Table 3 provides for reflect+ the number of times in which an instance was closed to optimality by $F_{RE}(P_{base})$ and $F_{RE}(P_{LP})$ (#U_1, U_2), by $F_{RE} + (21) - (23)$ (#U_3), and by the main algorithm's loop (#U_4). The best configuration we found for Reflect+ is as follows: For the standard instances (nb. var. plus 10 times nb. cons. lower than 1 000 000), we gave up to 60 seconds for U_1, up to 1200 for U_3, and the remaining time for U_4; for the other very large instances, we gave up to 3600 seconds for computing U_1, the remaining time, if any, to U_2 and then possibly to U_4.

Table 3: Comparison of reflect and reflect+ with the best algorithms from the BPP/CSP literature

set of instances	# inst.	Belov		VPSolver		reflect (F_{RE})		reflect+				
		#opt	sec	#opt	sec	#opt	sec	#opt	sec	$\#U_1, U_2$	$\#U_3$	$\#U_4$
Waescher	17	**17**	0.1	16	886.2	**17**	555.5	**17**	41.3	7	8	2
Hard28	28	**28**	7.5	**28**	33.0	**28**	19.0	**28**	4.2	0	23	5
Falkenauer U	80	**80**	0.0	**80**	0.1	**80**	0.0	**80**	0.1	79	1	0
Falkenauer T	80	**80**	56.9	**80**	0.4	**80**	0.3	**80**	1.0	10	70	0
Schwerin 1	100	**100**	1.0	**100**	0.3	**100**	0.3	**100**	0.1	100	0	0
Schwerin 2	100	**100**	1.3	**100**	0.3	**100**	0.2	**100**	0.1	100	0	0
Scholl 1	720	**720**	0.0	**720**	0.0	**720**	0.0	**720**	0.1	717	3	0
Scholl 2	480	**480**	0.3	479	107.7	**480**	9.6	**480**	2.8	475	5	0
Scholl 3	10	**10**	14.1	**10**	8.5	**10**	8.6	**10**	3.7	10	0	0
total/avg (1)	1615	**1615**	3.3	1613	42.1	**1615**	9.1	**1615**	1.5	1498	110	7
GI AA	60	**60**	2.8	56	453.8	**60**	179.8	**60**	11.7	60	0	0
GI AB	60	**60**	10.9	0	3600.0	0	3600.0	**60**	29.6	60	0	0
GI BA	60	**60**	2.8	57	491.6	**60**	380.1	**60**	16.4	59	1	0
GI BB	60	**60**	10.5	0	3600.0	0	3600.0	**60**	47.5	60	0	0
total/avg (2)	240	**240**	6.8	113	1968.4	120	1940.0	**240**	26.3	239	1	0
AI 200	50	**50**	90.6	**50**	105.8	**50**	45.5	**50**	8.5	1	48	1
AI 400	50	**45**	699.4	36	1430.5	21	2297.4	40	1205	0	30	10
ANI 200	50	**50**	144.2	49	119.5	**50**	67.2	**50**	49.3	0	0	50
ANI 400	50	1	3555.6	11	3170.2	10	3083.6	**17**	2703.9	0	0	17
total/avg (3)	200	146	1222.5	146	1206.5	131	1373.4	**157**	991.7	1	78	78
total/avg (1)+(2)+(3)	2055	2001	122.3	1872	372.6	1866	190.3	**2012**	100.7	1738	189	85

Table 3 shows that the behavior of reflect+ is very satisfactory and that the additional techniques help improving the results obtained by reflect alone. While U_1 and U_2 are effective for the first two benchmark sets, U_3 and U_4 are useful for the third difficult set. On average, reflect+ consistently outperforms VPSolver in terms of number of optimal solutions and time, especially on difficult instances (Waescher, GI AB and BB, and AI 400). When compared to Belov, reflect+ is less powerful on the AI 400 instances (where the issue is to find the good heuristic solution) but better for the ANI 400 instances (where it is difficult to raise the lower bound to the optimal value), and overall finds 11 more proven optimal solutions with a slightly smaller computational effort.

7.2 Results on the VSBPP

To test F_{RE}^{VS} and reflect$_{+}^{VS}$, we considered three VSBPP benchmark sets:

(4) Crainic: 3 sets used in Crainic et al. (2011), gathering instances from the VSBPP literature having up to 12 bin types and 1000 items, and being rather easy because of small c values;

(5) Hemmelmayr: 2 sets of instances used in Hemmelmayr et al. (2012) and derived from previous articles, containing up to 7 bin types and 2000 items, and having moderate difficulty;

(6) Belov: 4 sets of difficult instances proposed by Belov and Scheithauer (2002). Each set contains 50 instances with around 5000 items and bin capacity up to 10 000. The maximum number of different bins is 2, 4, 8, and 16 for, respectively, Belov 1, 2, 3, and 4. Belov 2 is a standard reference set also addressed by Alves and Valério de Carvalho (2008). The other three sets are interesting to study the impact of $|T|$.

Table 4 compares our methods with the best algorithms available in the VSBPP literature, namely: *Belov*, the branch-and-cut-and-price algorithm by Belov and Scheithauer (2002); *Alves*, the branch-and-cut-and-price by Alves and Valério de Carvalho (2008); *Hemmelmayr*, the combination of lower bounds and variable neighbourhood search by Hemmelmayr et al. (2012). We also present the results of arc-flow, that for the VSBPP was not computationally tested up to now. All methods were executed with a time limit of 3600 seconds on our machine, with the exception of Alves, executed on an Intel Core Duo 1.83GHz (passmark value 728), and Hemmelmayr, executed on an Intel Pentium D 940 3.20GHz (passmark value 710). We set the Reflect$^{VS}_{+}$ configuration as follows: For small instances ($n \leq 500$) we executed directly U_4; for the other instances, we gave up to 60 seconds for U_1, up to 60 seconds for U_2, and the remaining time to U_4.

Table 4: Comparison of F^{VS}_{RE} and reflect$^{VS}_{+}$ with the best algorithms from the VSBPP literature

set of instances	# inst.	Belov		Alves*		Hemmelmayr**		arc-flow		reflect (F^{VS}_{RE})		reflect$^{VS}_{+}$	
		#opt	sec	#opt	sec	#opt	sec	#opt	sec	#opt	sec	#opt	sec
Crainic 1	300	-	-	**300**	0.2	-	-	**300**	1.1	**300**	0.3	**300**	0.3
Crainic 2	60	-	-	-	-	-	-	**60**	3.4	**60**	2.2	**60**	2.5
Crainic 3	480	-	-	-	-	-	-	**480**	0.6	**480**	0.2	**480**	0.2
total/avg (4)	840	-	-	-	-	-	-	**840**	1.0	**840**	0.4	**840**	0.4
Hemmelmayr 1	150	-	-	-	-	78	13.4	**150**	2.7	**150**	1.2	**150**	1.1
Hemmelmayr 2	50	-	-	-	-	**50**	0.7	**50**	1.5	**50**	0.5	**50**	0.8
total/avg (5)	200	-	-	-	-	128	10.3	**200**	2.4	**200**	1.0	**200**	1.0
Belov 1	50	42	753.4	-	-	-	-	38	1230.6	48	258.5	**50**	6.1
Belov 2	50	38	1058.5	47	227.5	-	-	34	1679.7	41	874.3	**50**	4.1
Belov 3	50	37	1110.3	-	-	-	-	31	1673.3	44	786.0	**50**	1.6
Belov 4	50	39	1127.0	-	-	-	-	27	2009.2	35	1613.1	**50**	4.0
total/avg (6)	200	156	1012.3	-	-	-	-	130	1648.2	168	883.0	**200**	3.9
total/avg (4)+(5)+(6)	1240	-	-	-	-	-	-	1170	266.9	1208	142.9	**1240**	1.1

* executed on an Intel Core Duo 1.83GHz; ** executed on an Intel Pentium D 940 3.20GHz

Table 4 shows that pseudo-polynomial formulations are, in general, very effective for most of the VSBPP instances. They are sufficient to solve all instances of the first two benchmark sets in just a few seconds, finding comparable results with Alves on Crainic 1 and outmatching Hemmelmayr on the second benchmark set. The real challenge is on the third difficult set. For example, for Belov 2 arc-flow finds only 34 proven optima and is outmatched by the two methods based on cut and column generation by Belov (38 optima) and Alves (47 optima). A similar behavior can be observed for Belov 1, 3, and 4. Reflect is better than Belov on average but slightly worse than Alves, whereas reflect$^{VS}_{+}$ marks a strong improvement, solving in about 4 seconds all 200 instances (U_1 and U_2 found 194 proven optimal solutions and U_4 closed the 6 remaining instances). Overall, reflect$^{VS}_{+}$ closed all instances in a second on average. It is also interesting to notice that parameter $|T|$ does not influence reflect$^{VS}_{+}$, has a small impact on Belov algorithm, but affects the performance of both arc-flow (from 38 to 27 optima) and reflect (from 48 to 35 optima). This can be partially imputed to the number of variables in the models, which for reflect increases from an average value of 37 708 for Belov 1 (maximum $|T|$=2) to 52 248 for Belov 4 (maximum $|T| = 16$).

7.3 Results on the BPPIF

We used two BPPIF benchmark sets:

(7) bm-BPPIF: A set of 540 instances by Casazza and Ceselli (2016) having up to 1000 items (divided in ranges 20–100 and 150–1000), two types of bin capacity (*tight*, *loose*), and three range of item weights (*free*, *large*, *small*);

(8) fm-BPPIF: A copy of the previous set where the limit B on the number of bins has been replaced by a limit F on the number of items.

Table 5 compares our results with those of the branch-and-price by Casazza and Ceselli (2016), denoted as *Casazza*. All algorithms were run with a time limit of 3600 second, Casazza on an Intel Core2 Duo E6850 3.00 GHz (passmark 1951). Apart from #opt and sec, the table also provides for reflect^{IF} the number of instances solved to proven optimality by F_{RE}^{VS} with different $|T|$ values (#$|T|$=1, 2, 3) and by F_{AF}^{bm}/F_{AF}^{fm} (#AF).

Table 5: Comparison of new arc-flow formulations and reflect^{IF} with the existing BPPIF literature

variant	set of instances		# inst.	Casazza*		arc-flow (F_{AF}^{bm}/F_{AF}^{fm})		reflect^{IF}					
				#opt	sec	#opt	sec	#opt	sec	#$\lvert T\rvert$=1	#$\lvert T\rvert$=2	#$\lvert T\rvert$=3	#AF
	20-100	loose-free	30	**30**	2.8	**30**	1.2	**30**	0.6	29	1	0	0
	20-100	loose-large	30	**30**	1.8	**30**	3.3	**30**	1.4	0	30	0	0
	20-100	loose-small	30	**30**	18.1	**30**	35.0	**30**	1.0	30	0	0	0
	20-100	tight-free	30	**30**	4.3	**30**	9.3	**30**	1.9	11	19	0	0
	20-100	tight-large	30	**30**	1.5	**30**	4.1	**30**	1.5	0	30	0	0
bm	20-100	tight-small	30	**30**	3.5	**30**	7.8	**30**	1.6	30	0	0	0
	150-1000	loose-free	60	42	1591.8	**60**	6.9	**60**	3.4	60	0	0	0
	150-1000	loose-large	60	50	1205.7	**60**	17.5	**60**	7.3	0	60	0	0
	150-1000	loose-small	60	30	1950.5	47	1528.2	**60**	16.5	60	0	0	0
	150-1000	tight-free	60	44	1891.6	**60**	140.0	**60**	25.7	0	60	0	0
	150-1000	tight-large	60	49	1514.8	**60**	19.8	**60**	6.4	0	60	0	0
	150-1000	tight-small	60	48	1453.0	49	1176.0	**60**	28.5	60	0	0	0
total/avg (7)			540	443	803.3	516	324.3	**540**	10.2	280	260	0	0
	20-100	loose-free	30	**30**	2.1	**30**	6.6	**30**	1.3	26	4	0	0
	20-100	loose-large	30	**30**	2.0	**30**	6.1	**30**	1.2	0	30	0	0
	20-100	loose-small	30	**30**	0.3	**30**	9.6	**30**	1.1	30	0	0	0
	20-100	tight-free	30	**30**	85.7	28	328.8	**30**	28.7	0	13	11	6
	20-100	tight-large	30	**30**	81.4	**30**	55.5	**30**	11.1	0	0	18	12
fm	20-100	tight-small	30	**30**	12.4	**30**	193.9	**30**	3.4	15	14	0	1
	150-1000	loose-free	60	0	3600.0	**60**	24.9	**60**	6.1	60	0	0	0
	150-1000	loose-large	60	59	1044.6	**60**	42.7	**60**	5.4	0	60	0	0
	150-1000	loose-small	60	0	3600.0	**60**	12.2	**60**	5.4	60	0	0	0
	150-1000	tight-free	60	11	3382.0	11	3080.6	**60**	217.0	0	58	0	2
	150-1000	tight-large	60	39	1885.5	39	1694.6	**59	621.3	0	0	59	0
	150-1000	tight-small	60	0	3600.0	27	2453.5	**60**	80.6	60	0	0	0
total/avg (8)			540	289	1441.3	435	845.4	**539**	106.6	251	179	88	21
total/avg (7)+(8)			1080	732	1122.3	951	584.9	**1079**	58.4	531	439	88	21

* tested on an Intel Core2 Duo E6850 3.00GHz; ** remaining instance solved to proven optimality in 4858.7 sec

The results show that our algorithms are very effective. All instances with up to 100 items are easily solved by all methods. For the larger instances, our arc-flow formulations outmatch Casazza both in terms of #opt and sec on both problem variants. A good improvement is then obtained by reflect^{IF}, which solves to optimality all bm instances and all fm instances but one. Between the two variants, bm is the easiest as all instances are solved by reflect^{IF} in about 10 seconds on average, and by just invoking F_{RE}^{VS} with $|T| = 1$ or 2. The fm variant is harder and imposes a larger burden to all algorithms. The only fm instance that was unsolved to proven optimality in one hour could be solved by reflect^{IF} in about 80 minutes. In this way, optimal solutions have been produced for all benchmark instances.

8 Conclusions and future research

Thanks to the sharp improvement in the performance of mixed integer linear programming solvers, pseudo-polynomial models have recently become a useful tool for the solution of many combinatorial optimization problems. This has raised interesting opportunities for the development of new effective combinatorial techniques and for devising dedicated solution algorithms.

In our work, we studied pseudo-polynomial models for the classical bin packing problem (BPP) and cutting stock problem. We gave a complete overview of the dominance and equivalence relations among the main formulations. We then introduced a formulation that models a BPP instance on a reduced network where source-to-sink paths are replaced by pairs of colliding paths. We improved the computational performance of this formulation by using techniques based on column generation, dual cuts, and heuristics, and showed the easy replicability of the proposed methods by adapting them to two relevant BPP generalizations, namely, the variable sized BPP and the BPP with item fragmentation. Extensive computational results proved that our algorithms are very effective, improving upon the existing literature and providing optimal solutions for many benchmark instances that were previously unsolved.

Research on pseudo-polynomial models is currently of great interest, because it could help improving state-of-the-art results in many decision problems of practical interest. Relevant applications could arise in one-dimensional cutting and packing, for example in problems with loose bin capacities (Ceselli and Righini 2008), reusable leftovers (Arbib et al. 2002) or cardinality constraints (Sadykov and Vanderbeck 2013), and in two-dimensional cutting and packing, for example as master problems in primal decomposition methods (as in Côté et al. 2014 and Delorme et al. 2017b). But they could arise also in resource-constrained and/or precedence-constrained problems, in the capacitated vehicle routing area, as well as in single and multiple machine scheduling problems under different objective functions. In practice, in many of those combinatorial optimization problems where capacity play a central role. Multi-objective optimization could also be tackled, by imposing the different profits/costs on the arcs of the network and using path reconstruction algorithms to derive Pareto optimal solutions. The development of new advanced techniques for improving the existing models is also of great interest. For the BPP, our pseudo-polynomial models are particularly put under strain by instances with large bin capacities and a large number of items of small weight, and, in fact, instances with just 400 items remain unsolved despite the application of dozens of algorithms. Future research is thus envisaged, both to devise new applied models and improve the existing ones.

Appendix

In this appendix, we provide the proofs of the statements in the paper, as well as some additional material that is useful for their comprehension. To facilitate the reader, some problem definitions and technical notations given in the paper are also summarized here.

Appendix A Details for Lemma 1

For the sake of clarity, we provide in Algorithm 2 the procedure that we use for the decomposition into paths of a solution of the continuous relaxation of arc-flow. The algorithm receives in input a generic solution $\bar{x}_{de}$ of $L(F_{AF})$, that is, the linear programming relaxation of model (7)–(10) in which (10) is replaced by $x_{de} \geq 0$. Let P define a generic set of paths. For short, let p define both a path and the index of such path, for $p \in P$. Algorithm 2 selects an arc emanating from the source node 0 to initialize a path (steps 3 and 4), iteratively extends the path until it reaches the sink node c (steps 5–8), and sets the flow on the path as the minimum flow on the selected arcs (steps 9 and 10). Then, the path is added to P (step 11) and the process is iterated until all variables take value 0. The set P is finally returned. Clearly, the algorithm also works for integer F_{AF} solutions.

Algorithm 2 `DecomposeAF`

```
 1: Input: An $L(F_{AF})$ solution $\bar{x}_{de}$
 2: $P \leftarrow \emptyset$
 3: while $\exists$ an arc $(0, e)$ with $\bar{x}_{0e} > 0$ do
 4:     $p \leftarrow \emptyset$; $d \leftarrow 0$
 5:     while $d \neq c$ do
 6:         select the first arc $(d, e) \in \delta^+(d)$ with $\bar{x}_{de} > 0$ and add it to $p$
 7:         $d \leftarrow e$
 8:     end while
 9:     $\bar{z}_p \leftarrow \min_{(d,e)\in p}\{\bar{x}_{de}\}$
10:     for all $(d, e) \in p$ do $\bar{x}_{de} \leftarrow \bar{x}_{de} - \bar{z}_p$
11:     $P \leftarrow P \cup \{p\}$
12: end while
13: return $P$
```

Appendix B Proof of Lemma 2

LEMMA 2. *Any solution of one-cut or of its continuous relaxation can be decomposed into a set of binary trees.*

Proof. The proof is based on the procedure that we use to decompose the solution into trees, given in Algorithm 3. The algorithm receives in input a solution $\bar{y}_{pq}$ of $L(F_{OC})$, having objective function value $\bar{z}$. Let T define a set of binary trees. Let t define both a tree and the index of such tree, for $t \in T$. Each tree is formed by a set of nodes, each having at most two children, a left one and a right one. Intuitively, children nodes are created by a cut on the piece (entire piece or residual piece) corresponding to their parent node. Let us use $[p, q]$ to denote a cut on a piece of width p that produces a left child of size q (and a right child of size $p - q$).

Algorithm 3 requires a simple mapping M that associates each cut $[p, q]$ with an index m_{pq}. At step 3, it selects a cut $[c, q]$ with positive $\bar{y}_{cq}$ value, and then, at steps 4-6, it uses the cut to initialize tree t, the set of cuts $\mathcal{C}$ used to generate t, and an array $\mathcal{L}$ storing the leaves of the tree. As long as there is a cut on a piece of width p stored in $\mathcal{L}$, then the tree and the corresponding set of cuts are enlarged (steps 7–13). The array $\mathcal{L}$ is updated by removing the piece that was cut and inserting the two children. After the process of enlarging the tree terminates, the value $\bar{z}_t$ of the tree is set to a minimum considering the values of the selected cuts and the number of times they appear in the tree (step 14). The residual variables values are updated at step 15, and the process continues until all trees have been generated.

Algorithm 3 `DecomposeOC`

```
 1: Input: An $L(F_{OC})$ solution $\bar{y}_{pq}$ and a map $M$ that associates cut $[p,q]$ with index $m_{pq}$
 2: $T \leftarrow \emptyset$
 3: while $\exists$ a cut $[c,q]$ with $\bar{y}_{cq} > 0$ do
 4:     $\mathcal{L}[0,\dots,c] \leftarrow 0$            ▷ an array that keeps track of the final leafs of the tree
 5:     $\mathcal{L}[c] \leftarrow 1$; initialize $t$ with $c$            ▷ a tree starts with a unique leaf $c$
 6:     $\mathcal{C}[0,\dots,\max\{m_{pq}\}] \leftarrow 0$     ▷ an array that keeps track of all the cuts used in the tree
 7:     while $\exists$ a cut $[p,q]$ with $\mathcal{L}[p] > 0$ and $\bar{y}_{pq} > 0$ do
 8:         $\mathcal{L}[p] \leftarrow \mathcal{L}[p] - 1$
 9:         add $q$ as left child of $p$ in $t$ and $p - q$ as right child of $p$ in $t$
10:         $\mathcal{L}[q] \leftarrow \mathcal{L}[q] + 1$
11:         $\mathcal{L}[p-q] \leftarrow \mathcal{L}[p-q] + 1$
12:         $\mathcal{C}[m_{pq}] \leftarrow \mathcal{C}[m_{pq}] + 1$
13:     end while
14:     $\bar{z}_t \leftarrow \min_{[p,q]:\mathcal{C}[m_{pq}]>0}\{\bar{y}_{pq}/\mathcal{C}[m_{pq}]\}$
15:     for all $[p,q] : \mathcal{C}[m_{pq}] > 0$ do $\bar{y}_{pq} \leftarrow \bar{y}_{pq} - \bar{z}_t \cdot \mathcal{C}[m_{pq}]$
16:     $T \leftarrow T \cup \{t\}$
17: end while
18: return $T$
```

Algorithm 3 clearly creates binary trees, but, to prove that it effectively decomposes a one-cut solution, it must be shown that it also terminates with a situation in which all variables take value 0 and no uncut portions of the solution are left. This can be proved by showing that it is impossible to find a $q \in \mathcal{W} \cup \mathcal{R} \setminus \{c\}$ that satisfies

$$\sum_{r \in C(q)} \bar{y}_{qr} > 0 \;\wedge\; \sum_{p \in A(q)} \bar{y}_{pq} = 0 \;\wedge\; \sum_{p \in B(q)} \bar{y}_{p+q,p} = 0. \tag{38}$$

Indeed, if (38) was satisfied, then a piece of width q different than c has to be recut (condition 1), but q is neither produced by a left cut (condition 2), nor by a right cut (condition 3).

To this aim note that, after the variables' update at step 15, there are two possible cases for q:

- $\mathcal{L}(q) > 0$, so q belongs to the final leafs of the tree that was just built. According to step 7, this means that for such q there is no cut $[q, r]$ for which $\bar{y}_{qr} > 0$ (otherwise the while loop would have been repeated), and consequently the first condition of (38) is not satisfied for this q;

- $\mathcal{L}(q) = 0$, or, in other words, q does not belong to any of the final leafs of the tree. This means that, in tree t, the amount of $\bar{y}$ where q appears as a child ($\sum_{p \in A(q)} \bar{y}_{pq} + \sum_{p \in B(q)} \bar{y}_{p+q,p}$) is equal to the amount of $\bar{y}$ where q appears as a father ($\sum_{r \in C(q)} \bar{y}_{qr}$). This has the consequence of maintaining inequality (5) satisfied during the algorithmic iterations. Indeed, (5) was initially satisfied by $\bar{y}$

and now remains satisfied because step 15 removes the same amount of $\bar{y}$ from both sides of it. This implies that the three conditions in (38) cannot be simultaneously satisfied for this q.

□

We remark that the decomposition shown in Figure 2 of Section 3 has been obtained by a slight modification of Algorithm 3 that only creates right-sided trees. The modification, which is only needed for the proof of equivalence of F_{AF} with F_{OC}, is formally discussed in Remark 3 below.

Appendix C Proof of Proposition 1 and additional examples

We first observe two simple properties that will be used in the main proof.

Remark 2 *Any solution $\bar{x}_{de}$ of $L(F_{AF})$ can be transformed into an equivalent solution which can be decomposed into* left-aligned *paths, that is, paths in which loss arcs, if any, appear at the right of all item arcs.*

Recall that, according to Lemma 1, one can decompose $\bar{x}_{de}$ into paths. Remark 2 then follows from the fact that one can select paths that are not left-aligned, and move to the left as much as possible all their item arcs, until no more move can be made. Note that an easy way to obtain a left-aligned solution is to replace in $\mathcal{A}$ each loss arc $(d, d+1)$ with a new loss arc (d, c). These new loss arcs can only appear at the end of a path, thus imposing a left-alignment. Then, a solution with these alternative loss arcs can be easily mapped into an original $L(F_{AF})$ solution by replacing each selected (d, c) arc with a chain of $c - d$ unit-width loss arcs.

Remark 3 *Any solution $\bar{y}_{pq}$ of $L(F_{OC})$ can be transformed into an equivalent solution which can be decomposed into* right-sided *trees, that is, trees in which only right children are allowed to have their own children.*

According to Lemma 2, we know that $\bar{y}_{pq}$ can be decomposed into a set of trees. A decomposition into right-sided trees can be obtained by either (i) modifying the positions of the cuts, or (ii) modifying model (4)–(6) and Algorithm 3. Here we describe the second option, whose implementation is simpler. It consists of adding to model (4)–(6) the inequality

$$\sum_{r \in C(q)} y_{qr} \leq \sum_{p \in B(q)} y_{p+q,p} \quad q \in \mathcal{W} \cup \mathcal{R} \setminus \{c\} \tag{39}$$

and removing step 10 from Algorithm 3. Inequality (39) is a symmetry-breaking constraint that imposes to select, among the equivalent solutions, one in which the number of times q is recut (left-hand side of the inequality) is not greater than the number of times it appears as a right child (right-hand side), thus ensuring that all recuts are performed on a right child. Then, the modification in Algorithm 3 guarantees that no left child enters the candidate list for recutting (array $\mathcal{L}$), thus producing only right-sided trees.

Proposition 1. *F_{AF} is equivalent to F_{OC}.*

Proof. We first prove that F_{AF} is included in F_{OC}, and then that F_{OC} in included in F_{AF}.

Arc-flow is included in one-cut.

Let us consider a generic solution $\bar{x}_{de}$ of $L(F_{AF})$. According to Lemma 1, $\bar{x}_{de}$ can be decomposed into paths, and, according to Remark 2. we can suppose without loss of generality that these paths are left-aligned. We use $\bar{x}_{de}$ to build a feasible and same-cost solution $\bar{y}_{pq}$ of $L(F_{OC})$ by using Algorithm 4. The intuition behind the algorithm is that any item arc (d, e) becomes a cut on a piece of length $(c - d)$ to obtain a left piece of length $(e - d)$.

Algorithm 4 `Transform_AF_into_OC`

```
1: Input: A left aligned solution $\bar{x}_{de}$ of $L(F_{AF})$
2: for all $p \in \mathcal{R}, q \in \mathcal{W}, p > q$ do $\bar{y}_{pq} \leftarrow 0$
3: for all $(d, e) \in \mathcal{A}_{\mathrm{I}} : \bar{x}_{de} > 0, e < c$ do $\bar{y}_{c-d,e-d} \leftarrow \bar{x}_{de}$
4: return $\bar{y}$
```

We first note that, at the end of Algorithm 4 we obtain

$$\bar{y}_{c-d,e-d} \quad = \quad \bar{x}_{de} \quad (d, e) \in \mathcal{A}_{\mathrm{I}}. \tag{40}$$

We now show that $\bar{y}$ has the same cost of $\bar{x}$. By using (8) and recalling that no arc enters 0, we know that $\bar{z} = \sum_{(0,f)\in\delta^+(0)} \bar{x}_{0f}$. All paths in $\bar{x}_{de}$ are left-aligned and so they start with one of the m item arcs. Moreover, we can equivalently use w_i for $i = 1, \ldots, m$ or $q \in \mathcal{W}$ to express an item width. By using these considerations, in the given order, and applying (40) to $\bar{x}_{0q}$, we get

$$\bar{z} = \sum_{(0,f)\in\delta^+(0)} \bar{x}_{0f} = \sum_{i=1}^{m} \bar{x}_{0w_i} = \sum_{q\in\mathcal{W}} \bar{x}_{0q} = \sum_{q\in\mathcal{W}} \bar{y}_{c-0,q-0} = \sum_{q\in\mathcal{W}} \bar{y}_{cq},$$

which is equivalent to (4).

To prove the feasibility of $\bar{y}$, we first note that Algorithm 4 only creates non-negative variable values, so we only have to show that constraints (5) are satisfied for any $q \in \mathcal{W} \cup \mathcal{R} \setminus \{c\}$. We first focus on the case in which q is not an item width, that is, $q \in \mathcal{R} \setminus \mathcal{W} \setminus \{c\}$, which implies $L_q = 0$ and $A(q) = \emptyset$. We rewrite the flow conservation at intermediate nodes $e = 1, \ldots, c - 1$ in (8) as

$$\sum_{(d,e)\in\delta^-(e)} \bar{x}_{de} = \sum_{(e,f)\in\delta^+(e)} \bar{x}_{ef} = \sum_{\substack{(e,f)\in\delta^+(e),\\ f\neq c}} \bar{x}_{ef} + \sum_{(e,c)\in\delta^+(e)} \bar{x}_{ec} \geq \sum_{\substack{(e,f)\in\delta^+(e),\\ f\neq c}} \bar{x}_{ef}. \tag{41}$$

We apply (40) to the leftmost and rightmost elements of (41) and obtain

$$\sum_{d\in\mathcal{S}, e-d\in\mathcal{W}} \bar{y}_{c-d,e-d} \quad \geq \quad \sum_{e\in\mathcal{S}, f-e\in\mathcal{W}} \bar{y}_{c-e,f-e} \quad e = 1, \ldots, c-1.$$

This can be modified, by using the definitions of sets B ad C (see Section 2.3), as

$$\sum_{e-d\in B(c-e)} \bar{y}_{c-d,e-d} \quad \geq \quad \sum_{f-e\in C(c-e)} \bar{y}_{c-e,f-e} \quad e = 1, \ldots, c-1.$$

Then, by setting indices $q = c - e$, $p = e - d$, and $r = f - e$, we obtain

$$\sum_{p\in B(q)} \bar{y}_{p+q,p} \quad \geq \quad \sum_{r\in C(q)} \bar{y}_{qr} \quad q = 1, \ldots, c-1,$$

which proves that (5) is satisfied for any $q \in \mathcal{R} \setminus \mathcal{W} \setminus \{c\}$ (as $\mathcal{R} \setminus \mathcal{W} \setminus \{c\}$ is included in $\{1, \dots, c-1\}$). When $q \in \mathcal{W}$, we also need to take into account L_q and $A(q)$. We consider again (8), but this time focus on a vertex $e = c - w_i$. By replacing e with $c - w_i$, we get

$$\sum_{(d,c-w_i)\in\delta^-(c-w_i)} x_{d,c-w_i} = \sum_{(c-w_i,f)\in\delta^+(c-w_i)} x_{c-w_i,f} = \sum_{\substack{(c-w_i,f)\in\delta^+(c-w_i),\\ f\neq c}} x_{c-w_i,f} + x_{c-w_i,c},$$

which we rearrange as

$$x_{c-w_i,c} = \sum_{(d,c-w_i)\in\delta^-(c-w_i)} x_{d,c-w_i} - \sum_{\substack{(c-w_i,f)\in\delta^+(c-w_i),\\ f\neq c}} x_{c-w_i,f}. \tag{42}$$

We now consider demand constraints (9) and rewrite their left hand side as

$$\sum_{(d,d+w_i)\in\mathcal{A}} x_{d,d+w_i} = \sum_{\substack{(d,d+w_i)\in\mathcal{A},\\ d+w_i\neq c}} \bar{x}_{d,d+w_i} + \sum_{\substack{(d,d+w_i)\in\mathcal{A},\\ d+w_i=c}} \bar{x}_{d,d+w_i} = \sum_{\substack{(d,d+w_i)\in\mathcal{A},\\ d+w_i\neq c}} \bar{x}_{d,d+w_i} + x_{c-w_i,c}.$$

We embed (42) into this last equation, and then use the result to rewrite (9) as

$$\sum_{\substack{(d,d+w_i)\in\mathcal{A},\\ d+w_i\neq c}} \bar{x}_{d,d+w_i} + \sum_{(d,c-w_i)\in\delta^-(c-w_i)} \bar{x}_{d,c-w_i} - \sum_{\substack{(c-w_i,f)\in\delta^+(c-w_i),\\ f\neq c}} \bar{x}_{c-w_i,f} \geq d_i. \tag{43}$$

We apply (40) to all members of (43) ($\sum_{(d,d+w_i)\in\mathcal{A},d+w_i\neq c} \bar{x}_{d,d+w_i} = \sum_{d\in\mathcal{S},w_i\in\mathcal{W}} \bar{y}_{c-d,w_i}$, $\sum_{(d,c-w_i)\in\delta^-(c-w_i)} \bar{x}_{d,c-w_i} = \sum_{d\in\mathcal{S},c-w_i-d\in\mathcal{W}} \bar{y}_{c-d,c-w_i-d}$, and $\sum_{(c-w_i,f)\in\delta^+(c-w_i),f\neq c} \bar{x}_{c-w_i,f} = \sum_{c-w_i\in\mathcal{S},f-(c-w_i)\in\mathcal{W}} \bar{y}_{w_i,f-(c-w_i)}$) and rewrite (43) as

$$\sum_{d\in\mathcal{S},w_i\in\mathcal{W}} \bar{y}_{c-d,w_i} + \sum_{d\in\mathcal{S},c-w_i-d\in\mathcal{W}} \bar{y}_{c-d,c-w_i-d} - \sum_{c-w_i\in\mathcal{S},f-(c-w_i)\in\mathcal{W}} \bar{y}_{w_i,f-(c-w_i)} \geq d_i. \tag{44}$$

We use the definitions of sets A, B, and C and replace d_i with L_{w_i} in (44), so we obtain

$$\sum_{c-e\in A(w_i)} \bar{y}_{c-e,w_i} + \sum_{c-w_i-d\in B(w_i)} \bar{y}_{c-d,c-w_i-d} - \sum_{f-(c-w_i)\in C(w_i)} \bar{y}_{w_i,f-(c-w_i)} \geq L_{w_i}.$$

We use index $q \in \mathcal{W}$ instead of w_i for $i = 1, \dots, m$ and rewrite the variables indices, so we get

$$\sum_{p\in A(q)} \bar{y}_{pq} + \sum_{p\in B(q)} \bar{y}_{p+q,p} - \sum_{r\in C(q)} \bar{y}_{qr} \geq L_q$$

which proves that (5) is also feasible for any $q \in \mathcal{W}$ and concludes the first part of the proof.

One-cut is included in arc-flow.

Let us consider a solution $\bar{y}_{pq}$ of $L(F_{OC})$. From Lemma 2 we know that $\bar{y}_{pq}$ can be decomposed into a set of trees, and from Remark 3 we can suppose without loss of generality that $\bar{y}_{pq}$ accomplishes with (39) and can be decomposed into a set of right-sided trees. We use $\bar{y}_{pq}$ to build a feasible and same-cost solution $\bar{x}_{de}$ of $L(F_{AF})$ by using Algorithm 5. The algorithm considers the alternative loss arcs (d, c) that we introduced in Remark 2. The intuition behind Algorithm 5 is that each cut becomes either a

Algorithm 5 `Transform_OC_into_AF`

```
1: Input: A solution $\bar{y}_{pq}$ of $L(F_{OC})$ + (39)
2: for all $(d, e) \in \mathcal{A}$ do $\bar{x}_{de} \leftarrow 0$
3: for all $p \in \mathcal{R}, q \in \mathcal{W}, p > q : \bar{y}_{pq} > 0$ do
4:     $\bar{x}_{c-p,c-p+q} \leftarrow \bar{x}_{c-p,c-p+q} + \bar{y}_{pq}$
5:     $\bar{x}_{c-p+q,c} \leftarrow \bar{x}_{c-p+q,c} + \bar{y}_{pq}$
6:     if $p \neq c$ then $\bar{x}_{c-p,c} \leftarrow \bar{x}_{c-p,c} - \bar{y}_{pq}$ end-if
7: end-for
8: return $\bar{x}$
```

path that increases the flow on two arcs or a cycle that decreases the flow on one arc and increases it on two other arcs (two clarifying examples are provided at the end of the proof).

We first derive two equations that are useful for the proof. Algorithm 5 increases (steps 4 and 5) or decreases (step 6) the $\bar{x}$ values on the basis of the cuts selected in the solution of $L(F_{OC})$ + (39). Step 4 is invoked at most once for each pair of p and q values, thus leading to

$$\bar{x}_{c-p,c-p+q} \quad = \quad \bar{y}_{pq} \quad p \in \mathcal{R}, q \in \mathcal{W}, p > q. \tag{45}$$

Steps 5 and 6 may instead modify multiple times the same variable $x_{c-q,c}$ for a given value of q. By considering all cuts $[r, s]$ for which $r - s = q$, step 5 leads to $\bar{x}_{c-q,c} \leftarrow \sum_{r \in \mathcal{R}, s \in \mathcal{W}, r > s : r-s=q} \bar{y}_{rs} = \sum_{s+q \in \mathcal{R}, s \in \mathcal{W}} \bar{y}_{s+q,s}$. By considering all cuts $[r, s]$ for which $r = q$, step 6 leads instead to $\bar{x}_{c-q,c} \leftarrow -\sum_{r \in \mathcal{W}, s \in \mathcal{R}, r > s : r=q} \bar{y}_{rs} = \sum_{q \in \mathcal{W}, s \in \mathcal{R}, q > s} \bar{y}_{qs}$. By summing these two components and recalling that $B(q) = \{p \in \mathcal{W} : p + q \in \mathcal{R}\}$ and $C(q) = \{p \in \mathcal{W} : p < q\}$, we get

$$\bar{x}_{c-q,c} \quad = \quad \sum_{p \in B(q)} \bar{y}_{p+q,p} - \sum_{r \in C(q)} \bar{y}_{qr} \quad q \in \mathcal{R} \setminus \{c\}. \tag{46}$$

To show that $\bar{x}$ has the same cost of $\bar{y}$, it is enough to apply (45) to (4), obtaining

$$z = \sum_{q \in \mathcal{W}} \bar{y}_{cq} = \sum_{q \in \mathcal{W}} \bar{x}_{c-c,c-c+q} = \sum_{q \in \mathcal{W}} \bar{x}_{0,q} = \sum_{(0,e) \in \delta^+(0)} \bar{x}_{0,e}$$

which is equivalent to (7).

We now show that constraints (8) remain satisfied during any iteration of Algorithm 5. This can be intuitively noticed by the example presented after this proof (see Figure C1). Formally, constraints (8) are satisfied after the initialization step as all flows take value 0. Then, let us first consider the flow variation involved by $\bar{y}_{pq} > 0$ having $p \neq c$. At step 4, the flow leaving $c - p$ is increased by $\bar{y}_{pq}$, but it is then reduced by the same amount at step 6, resulting in no flow variation. The same observation holds for the flow entering c at steps 5 and 6. The flows entering and leaving $c - p + q$ are balanced at steps 4 and 5. Now, let us consider the flow variations involved by a $\bar{y}_{cq} > 0$. Again, the flows entering and leaving q are increased by the same amount at steps 4 and 5. At step 4, the flow leaving 0 is increased by $\bar{y}_{cq}$, which is the same amount added to the flow entering c at step 5. Consequently, constraints (8) are satisfied by the built $\bar{x}$ values.

To prove that demand constraints (9) are also satisfied, we start by considering (5) for $q \in \mathcal{W}$ as

$$\sum_{p \in A(q)} \bar{y}_{pq} + \sum_{p \in B(q)} \bar{y}_{p+q,p} - \sum_{r \in C(q)} \bar{y}_{qr} \geq L_q \quad q \in \mathcal{W},$$

which, by applying (45) and (46), can be modified into

$$\sum_{\substack{(c-p,c-p+q)\in\mathcal{A}',\\ c-p+q\neq c}} \bar{x}_{c-p,c-p+q} + \bar{x}_{c-q,c} \geq L_q \quad q \in \mathcal{W}.$$

We now set $w_i = q$ and $e = c - p$, replace L_{w_i} with d_i, and obtain the desired result

$$\sum_{\substack{(e,e+w_i)\in\mathcal{A},\\ e+w_i\neq c}} \bar{x}_{e,e+w_i} + \bar{x}_{c-w_i,c} = \sum_{(e,e+w_i)\in\mathcal{A}} \bar{x}_{e,e+w_i} \quad \geq \quad d_i \quad i = 1,\dots,m.$$

For what concerns non-negativity, this may be an issue only for variables $x_{c-q,c}$, the only ones affected by step 6. We can notice, however, that the right-hand side of (46) is forced to be non-negative by the additional constraint (39) that we imposed, and thus $\bar{x}_{c-q,c} \geq 0$ holds (an example of a solution that does not satisfy (39) and leads to a negative $\bar{x}$ value is shown in Figure C2). This shows that $\bar{x}$ is a feasible $L(F_{AF})$ solution and concludes the proof.

$\square$

For the sake of clarity, we provide two examples that clarify some details of the proof. In Figure C1, we depict the four steps performed by Algorithm 5 when applied to the $\bar{y}_{pq}$ solution of Example 1 discussed in Lemma 2. The solution consists of four cuts ($\bar{y}_{11,7} = 1$, $\bar{y}_{11,4} = 1/3$, $\bar{y}_{7,3} = 1/3$, and $\bar{y}_{4,3} = 2/3$) and so requires four iterations, shown in the figure from top to bottom. Suppose the algorithm selects the cuts in the order above. At iteration 1, it selects $[11, 7]$ and sets $\bar{x}_{0,7} = \bar{x}_{7,11} = 1$, obtaining the first path. At iteration 2, it selects $[11, 4]$, thus setting $\bar{x}_{0,4} = \bar{x}_{4,11} = 1/3$ and creating the second path. At iteration 3, it processes $[7, 3]$, and, because $p = 7 \neq c$, it sets $\bar{x}_{4,7} = 1/3$, increases $\bar{x}_{7,11}$ to 4/3, and decreases $\bar{x}_{4,11}$ to 0, thus moving the precedent flow on (4,11) to (4,7) and (7,11). At the last iteration, it selects $[4, 3]$, thus imposing $\bar{x}_{7,10} = \bar{x}_{10,11} = 2/3$ and $\bar{x}_{7,11} = 2/3$, moving part of the flow on (7,11) to (7,10) and (10,11).

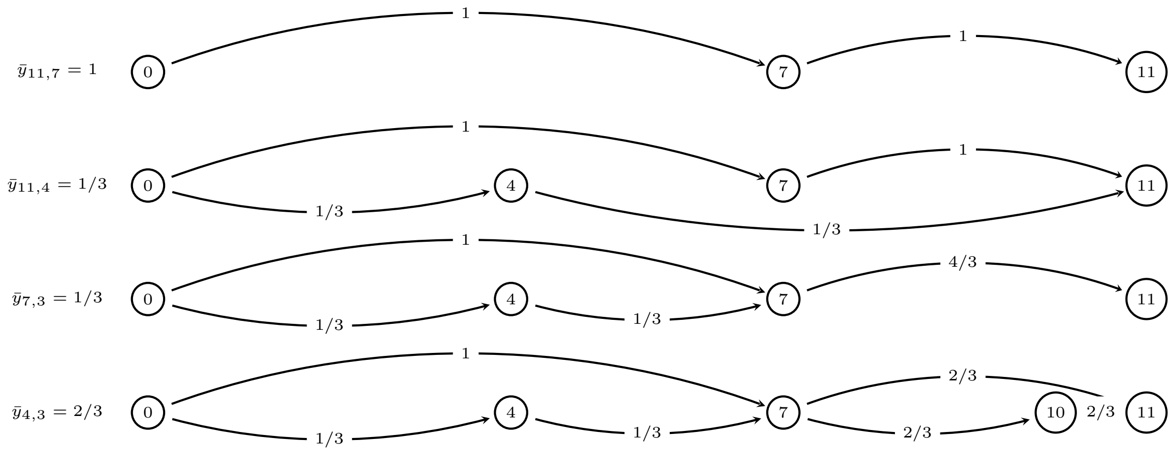


Figure C1: Construction of an $L(F_{AF})$ solution of Example 1 through Algorithm 5. Each iteration, from top to bottom, processes the cut associated with the $\bar{y}$ variable given on the left

The second example that we present in Figure C2 shows that, without the additional constraint (39), negative flows could appear in the solution produced by Algorithm 5. Indeed, let us consider an optimal $L(F_{OC})$ solution of Example 1 having value 4/3 and consisting of $\bar{y}_{11,7} = 4/3$, $\bar{y}_{7,3} = 1/3$, and

Algorithm 6 `Transform_DP_into_AF`

```
1: Input: A solution $\bar{\varphi}_{j,d,j+1,e}$ of $L(F_{DP})$
2: for all $(d,e) \in \mathcal{A}_{\mathrm{I}}$ do $\bar{x}_{de} \leftarrow \sum_{j=1}^{n} \bar{\varphi}_{j,d,j+1,e}$
3: for all $(d,d+1) \in \mathcal{A}_{\ell}$ do $\bar{x}_{d,d+1} \leftarrow \sum_{e=1}^{d} \bar{\varphi}_{n,e,n+1,c}$
4: return $\bar{x}$
```

$\bar{y}_{4,3} = 2/3$. The three iterations, one per cut, produced by Algorithm 5 are graphically depicted in the figure. It can be noticed that $\bar{x}_{4,11} = -1/3$, which, intuitively, corresponds to a cycle in the flow decomposition.

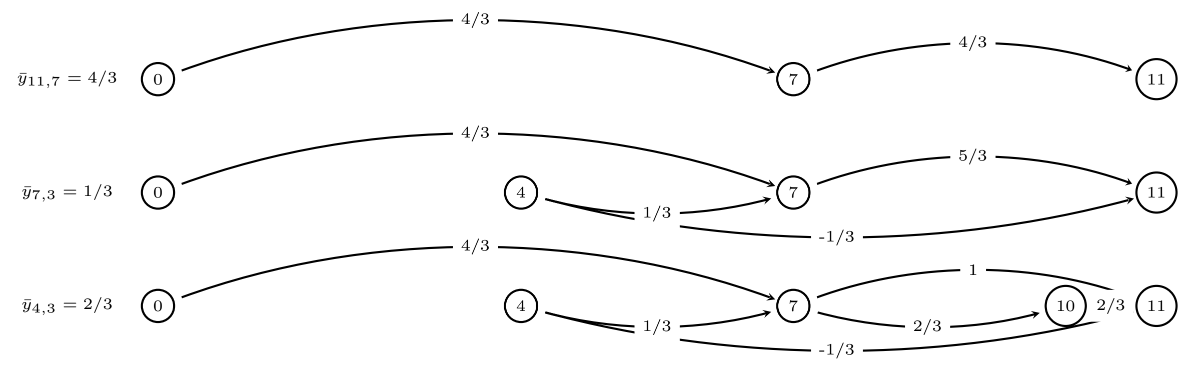


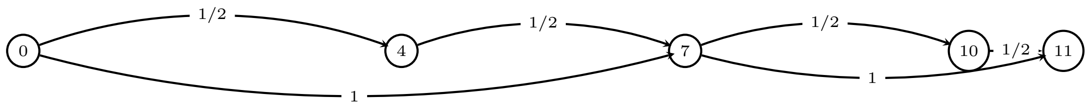
Figure C2: An infeasible $L(F_{AF})$ solution of Example 1 with a negative flow (cycle), obtained by executing Algorithm 5 on an input $L(F_{OC})$ solution not satisfying simmetry-breaking constraints (39)

Appendix D Proof of Proposition 2

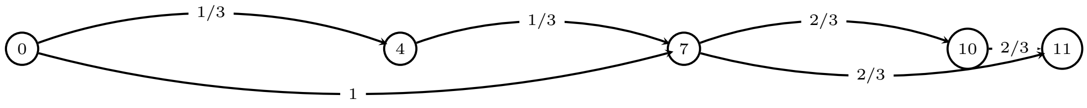
Proposition 2. *F_{DP} dominates F_{AF} (and hence F_{OC}).*

Proof. We first prove that F_{DP} is included in F_{AF}, and then give an example that shows that F_{AF} is not included in F_{DP}. For the first part, we make use of Algorithm 6. The algorithm takes in input an optimal $L(F_{DP})$ solution $\bar{\varphi}_{j,d,j+1,e}$ and uses it to create a feasible $L(F_{DP})$ solution $\bar{x}_{de}$. The idea, whose hint is in Delorme et al. (2016), is to vertically "shrink" all states with the same partial bin filling into a single node, and merge all arcs entering (resp. emanating) from the node into a unique entering (resp. emanating) arc. Consider again Example 1 of Section 2.2 and the optimal $L(F_{DP})$ solution depicted in Figure 3: By applying Algorithm 6, we obtain the feasible $L(F_{AF})$ solution shown in Figure D3-(a). Formally, (i) the variables associated to item arcs in F_{AF} are obtained in step 2 and become equal to the sum of the values of the merged variables from F_{DP}, and (ii) the variables associated to the loss arcs $(d, d+1)$ in F_{AF} are derived in step 3 by mapping the final arcs of F_{DP} (which, we recall, connect any node (n, d) with the dummy node $(n+1, c)$). The resulting $\bar{x}_{de}$ values satisfy constraints (8) because $\bar{\varphi}_{j,d,j+1,e}$ satisfy (12), and also constraints (9) because $\bar{\varphi}_{j,d,j+1,e}$ satisfy (13). This proves that F_{DP} is included in F_{AF}.

The same example can be used to show that F_{AF} is not included in F_{DP}. The optimal $L(F_{AF})$ solution discussed in Section 3 is graphically depicted in Figure D3-(b) and has value $4/3 < 3/2$. The reason of this AF discrepancy is that the partial filling of value 7 can be created by either the use of item 7 or the combined used of items 4 and 3. Consequently, the path (0,4,7,10,11) cannot be obtained in F_{DP} but can be easily produced by F_{AF}.



(a) A feasible $L(F_{AF})$ solution of value 3/2 obtained by Algorithm 6 on an optimal $L(F_{DP})$ solution



(b) An optimal $L(F_{AF})$ solution of value 4/3

Figure D3: Two fractional solutions of Example 1 that show that F_{AF} is not included in F_{DP}

□

Appendix E Proof of Proposition 3

Proposition 3. *F_{DP} is equivalent to F_{PR}.*

Proof. We first prove that F_{PR} is included in F_{DP}, and then that F_{DP} in included in F_{PR}. For the first part, we use the same idea of Lemma 1 and decompose an $L(F_{DP})$ solution into a set of paths. Then, we show that for each path p in F_{DP}, there is a pattern p' (i.e., a column) in F_{PR}. The idea is shown in Figure E4: Each vertical stage for p corresponds to a binary $a_{jp'}$ value of p'. If at stage j the arc in the path is vertical (i.e., $\varphi_{j,d,j+1,d}$ is selected), then $a_{jp'} = 0$. If instead the arc is diagonal (i.e., $\varphi_{j,d,j+1,d+w_j}$ is selected), then $a_{jp'} = 1$.

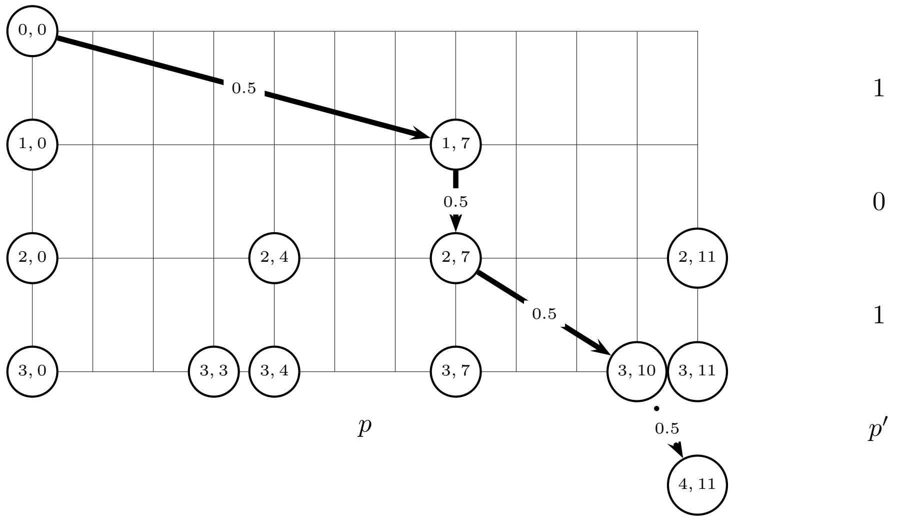


Figure E4: Transforming a path p of an $L(F_{DP})$ solution (left) into a pattern p' for $L(F_{PR})$ (right)

As F_{DP} is dedicated to the BPP, the transformation produces binary columns. It is possible to recreate afterwards integer columns for the CSP by summing the binary values $a_{jp'}$ associated with the same item type. As constraints (13) are satisfied in the solution of $L(F_{DP})$, they are also satisfied in the transformed patterns of $L(F_{PR})$. The reverse process is very similar: A BPP pattern of $L(F_{PR})$ can be easily mapped in a pattern for the CSP and then transformed into a path for $L(F_{DP})$. Any path created in this way satisfies flow conservation constraints (12). Constraints (13) are also satisfied, because demand constraints are satisfied in the $L(F_{PR})$ solution. Remark that, to maintain the " = " in (13) (instead of a " $\geq$ ") one should possibly have to remove some redundant item copies from the solution. □

Appendix F Proofs of Propositions 4 and 5

PROPOSITION 4. *F_{RE} models the CSP.*

Proof. Consider a generic solution of a CSP instance. We show that any pattern (bin filling) p in the solution can be transformed in reflect into a pair of paths that start in 0 and collide at a given vertex. To this aim, we sort items in p by non-increasing weight and divide them into two subsets, I_1 and I_2. We start by filling I_1 with the items in the computed order until their total width is either greater than or equal to $c/2$ or until there are no more items in p. We insert the rest of the items, if any, in I_2. For simplicity, suppose that items are numbered from 1 to $|I_1|$ in I_1 and from $|I_1| + 1$ to $|I_1| + |I_2|$ in I_2. We can distinguish three cases:

(1) $\sum_{i \in I_1} w_i < c/2$, so I_1 has not been filled up to $c/2$ and thus I_2 is empty. The pattern can be represented by a pair of paths colliding in $c/2$: (i) a path p_1 made by standard arcs corresponding to the items in I_1 (namely, $(0, w_1, s)$, $(w_1, w_1 + w_2, s)$, $\dots$), then loss arcs up to $c/2$, and then the reflected arc $(c/2, c/2, r)$; and (ii) a path p_2 containing only loss arcs from 0 to $c/2$.

(2) $\sum_{i \in I_1} w_i = c/2$. The pattern can be created by a pair of paths in which: (i) path p_1 is formed by standard arcs associated with the items in I_1, so ending in $c/2$, and a last arc $(c/2, c/2, r)$; and (ii) path p_2 is formed by standard arcs associated with the items in I_2 (if any) and possibly concluded by loss arcs until $c/2$ is reached. Once more the two paths collide in $c/2$.

(3) $\sum_{i \in I_1} w_i > c/2$. In this case the total width of the items in I_1 exceeds $c/2$, and hence: (i) path p_1 is formed by standard arcs associated with the first $|I_1| - 1$ items and a last reflected arc $(\bar{d}, \bar{e}, r)$, where $\bar{d} = \sum_{i=1,\dots,|I_1|-1} w_i$ and $\bar{e} = c - \bar{d} - w_{|I_1|}$; and (ii) path p_2 is formed by standard arcs associated with the items in I_2 (if any), possibly followed by loss arcs until $\bar{e}$ is reached. The pattern is then represented by the pair of paths p_1 and p_2, which collide in $\bar{e}$.

The opposite procedure also holds, as any pair of colliding paths in reflect can be transformed into a feasible pattern, and this concludes the proof. □

PROPOSITION 5. *Any feasible CSP pattern can be represented in F_{RE} by a pair of colliding paths whose reflected arc (d, e, r) has $d \leq e$.*

Proof. Proof We need to prove that reflect can model any CSP pattern p by using a reflected arc whose tail is at the left of its head. In other words, this means that at least half of the item width modeled by the reflected arc is packed before reflection. Once again, we sort items in p by non-increasing weight, divide them into I_1 and I_2, and consider the three cases in the proof of Proposition 4 above. The statement is clearly satisfied for cases (1) and (2), where the reflected arc is $(c/2, c/2, r)$. Let us focus on case (3) and consider $d > e$. We obtain an equivalent pair of paths that satisfies the statement by modifying the way in which we fill I_1 and I_2. We fill I_1 with the first $|I_1| - 1$ items as before, but then insert the item $|I_1|$ in I_2. We then scan items one at a time in the given order. Let j define the index of the current item. We insert j in I_1 if at least half of its width fits within $c/2$, namely, if $w_j/2 + \sum_{i \in I_1} w_i \leq c/2$ holds, as this would correspond to either a standard arc or a reflected arc accomplishing with $d \leq e$.

Otherwise, we insert it in I_2. We proceed with all items. Let now j be the item that, either packed in I_1 or I_2, would require a reflection, if any. One of the two possible reflections satisfies the statement, because otherwise we would have $w_j/2 + \sum_{i \in I_1} w_i > c/2$ and $w_j/2 + \sum_{i \in I_2} w_i > c/2$. But that would imply $w_j + \sum_{i \in I_1 \cup I_2} w_i = \sum_{i \in p} w_i > c$, which is impossible as p is a feasible pattern.

□

Appendix G Algorithms for reflect

In Algorithm 7, we detail how to build the multigraph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ that is at the basis of F_{RE}. Recall that the set of arcs is partitioned into standard arcs $(d, e, s) \in \mathcal{A}_s$ and reflected arcs $(d, e, r) \in \mathcal{A}_r$. The algorithm uses an array M to keep track of the possible arc tails. For a given item type, it also uses an array H to keep track of the possible arc tails that were already processed, to avoid unnecessary operations. For each item type i (step 5) and for each item j of type i (step 7), it scans all possible tails l that were not already processed (steps 8–9), and attempts to create an arc from l to $l + w_j$. In case of a standard arc (step 11), $(l, l + w_j, s)$ is stored in $\mathcal{A}_s$ and $l + w_j$ becomes both a vertex and a possible tail (steps 12–14). In case of a reflected arc not removed by the reduction procedure in Proposition 5 (step 15), $(l, c - (l + w_i), r)$ is stored in $\mathcal{A}_r$ and $c - (l + w_i)$ becomes a vertex but not a possible tail (steps 16–17). Finally, vertex $c/2$ is added to $\mathcal{V}$, a loss arc between each pair of consecutive vertices is created, and arc $(c/2, c/2, r)$ is added to $\mathcal{A}_r$ (steps 23–25).

Algorithm 7 `Create_reflect_multigraph`

```
 1: Input: $c$: Bin capacity, $m$: Number of item types, $w$: Item widths, $d$: Item demands
 2: $\mathcal{A}_s \leftarrow \emptyset$ ; $\mathcal{A}_r \leftarrow \emptyset$ ; $\mathcal{V} \leftarrow \emptyset$
 3: $M[0 \dots c/2] \leftarrow 0$                         ▷ an array that keeps track of the possible tails
 4: $M[0] \leftarrow 1$                                    ▷ node 0 is a possible tail
 5: for $i = 1$ to $m$ do
 6:     $H[0 \dots c/2] \leftarrow 0$             ▷ an array that keeps track of the tails already processed
 7:     for $j = 1$ to $d_i$ do
 8:         for $l = c/2 - 1$ down to $0$ do
 9:             if $H[l] = 0$ and $M[l] = 1$ then      ▷ if $l$ is a possible tail and was not processed yet
10:                 $H[l] = 1$                                     ▷ $l$ is now processed
11:                 if $l + w_i \le c/2$ then                      ▷ if the arc is standard
12:                     $\mathcal{A}_s \leftarrow \mathcal{A}_s \cup (l, l + w_i, s)$
13:                     $\mathcal{V} \leftarrow \mathcal{V} \cup \{(l + w_i)\}$
14:                     $M[l + w_i] \leftarrow 1$
15:                 else if $l \le c - (l + w_i)$ then     ▷ if the arc is reflected + reduction procedure
16:                     $\mathcal{A}_r \leftarrow \mathcal{A}_r \cup (l, c - (l + w_i), r)$
17:                     $\mathcal{V} \leftarrow \mathcal{V} \cup \{(c - (l + w_i))\}$
18:                 end if
19:             end if
20:         end for
21:     end for
22: end for
23: $\mathcal{V} \leftarrow \mathcal{V} \cup \{c/2\}$
24: for $i \in \mathcal{V}$ do $\mathcal{A}_s \leftarrow \mathcal{A}_s \cup \{(i, j, s)\colon\ j = \min(l \in \mathcal{V} : l > i)\}$      ▷ loss arcs
25: $\mathcal{A}_r \leftarrow \mathcal{A}_r \cup (c/2, c/2, r)$                ▷ allow paths that collide in $c/2$
26: return $\mathcal{V}, \mathcal{A}_s, \mathcal{A}_r$
```

Note that Algorithm 7 would provide a valid multigraph to F_{RE} even if step 15 were replaced by a simple "**else**". Indeed, that would result in a larger graph, not accomplishing with the reduction of Proposition 5, but having all required arcs to model a CSP instance. Note also that the algorithm works for even values of c. In case c is an odd number, an easy adaptation is to double the capacity of the bin and the width of all item types. It is worth noticing that this operation does not affect the number of possible item widths combinations, and thus does not increase the number of variables and constraints in F_{RE}. Otherwise, an additional dummy node R can be created and taken into explicit consideration in the algorithm.

Algorithm 8 `Reconstruct_reflect_solution`

```
 1: Input: A solution $\bar{\xi}_{de\kappa}$ of $L(F_{RE})$
 2: $P \leftarrow \emptyset$; $S[0,\dots,c/2] \leftarrow \emptyset$; $R[0,\dots,c/2] \leftarrow \emptyset$
 3: while $\exists$ an arc $(0,e,\kappa)$ with $\bar{\xi}_{0e\kappa} > 0$ do        ▷ build the paths and store them in $R$ and $S$
 4:     $p \leftarrow \emptyset$; $d \leftarrow 0$
 5:     while $\exists$ an arc $(d,e,\kappa)$ with $\bar{\xi}_{de\kappa} > 0$ and $\nexists(d,e,r) \in p$ do
 6:         select the first arc $(d,e,\kappa) \in \delta^+(d)$ with $\bar{\xi}_{de\kappa} > 0$ and add it to $p$
 7:         $d \leftarrow e$
 8:     end while
 9:     $\bar{z}_p \leftarrow \min_{(d,e,\kappa)\in p}\{\bar{\xi}_{de\kappa}\}$
10:     for all $(d,e,\kappa) \in p$, $\bar{\xi}_{de\kappa} \leftarrow \bar{\xi}_{de\kappa} - \bar{z}_p$
11:     if $\exists(d,e,r) \in p$ then $R[d] \leftarrow R[d] \cup \{p\}$ else $S[d] \leftarrow S[d] \cup \{p\}$
12: end do
13: for $d = 1$ to $c/2$ do        ▷ match the pairs of colliding paths and store them in $P$
14:     while $\exists p_1 \in R[d]$ with $\bar{z}_{p1} > 0$ do
15:         select the first path $p_2 \in S[d]$ with $\bar{z}_{p2} > 0$
16:         $p \leftarrow p_1 \cup p_2$
17:         $\bar{z}_p \leftarrow \min(\bar{z}_{p1}, \bar{z}_{p2})$
18:         $\bar{z}_{p1} \leftarrow \bar{z}_{p1} - \bar{z}_p$; $\bar{z}_{p2} \leftarrow \bar{z}_{p2} - \bar{z}_p$
19:         $P \leftarrow P \cup p$
20:     end do
21: end do
22: return $P$
```

Algorithm 8 shows how to reconstruct a solution after F_{RE} has been solved. The algorithm uses a set P of colliding paths, as well as two arrays of sets of standard and reflected paths, S and R, respectively. After initialization, it creates a path from 0 to its colliding node d (steps 3-12), and stores it either in $R[d]$ if it contains a reflected arc, or in $S[d]$ otherwise (step 11). The process is iterated until no more paths can be found. Then, for each possible colliding node d (step 13), the algorithm selects two paths from $R[d]$ and $S[d]$ (steps 14–15) and merge them to reconstruct a pair p (steps 16–19). Such merging is ensured by constraints (17), that guarantee that each selected reflected path can be matched with a selected standard path. For the sake of completeness, we remark that such merging is guaranteed if all item types i have $w_i < c$, accomplishing with our initial assumption in Section 2.1. Otherwise, there could be a selected reflected arc $(0,0,r)$ corresponding to an item of size c that could not be matched with any standard path. Clearly, any item type having $w_i = c$ can be removed from the instance in a preprocessing phase.

Appendix H Algorithm for VSBPP

In Algorithm 9, we detail how to build the multigraph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ used within F_{RE}^{VS} to solve the VSBPP. We recall that c stands for the maximum capacity of a bin, and that the arc set $\mathcal{A}$ is partitioned into the set $\mathcal{A}_s$ of standard arcs and the set $\mathcal{A}_r$ of reflected arcs. Moreover, T is the set of bin types, and $\mathcal{A}_r = \cup_{t \in T} \mathcal{A}_r(t)$, where $\mathcal{A}_r(t)$ is the set of arcs that have been reflected at $c_t/2$. Algorithm 9 is quite similar to Algorithm 7. It makes use of an array M to store the possible arcs tails, and of an array H to keep track of the possible arc tails that were already processed. After initialization, it considers in turn each item type i, each item j, and each possible tail l that was not already processed (steps 5–9). When processing l, it first attempts to create a standard arc from l to $l + w_j$ (steps 11–15), and then attempts to create reflected arcs for all possible bin types in T (steps 16–21). It then concludes by creating all required sets at steps 26–28.

Algorithm 9 `Create_reflect_multigraph_VSBPP`

```
 1: Input: $c_t$: Bin capacities, $c$: Largest bin capacity, $m$: Number of item types, $w$: Item widths, $d$:
    Item demands
 2: $\mathcal{V} \leftarrow \emptyset$; $\mathcal{A}_s \leftarrow \emptyset$; for $t \in T$ do $\mathcal{A}_r(t) \leftarrow \emptyset$
 3: $M[0, \dots, c/2] \leftarrow 0$                          ▷ an array that keeps track of the possible tails
 4: $M[0] \leftarrow 1$                                       ▷ node 0 is a possible tail
 5: for $i = 1$ to $m$ do
 6:     $H[0, \dots, c/2] \leftarrow 0$                      ▷ an array that keeps track of the tails already processed
 7:     for $j = 1$ to $d_i$ do
 8:         for $l = c/2 - 1$ down to 0 do
 9:             if $H[l] = 0$ and $M[l] = 1$ then            ▷ if $l$ is a possible tail and was not processed yet
10:                 $H[l] = 1$                                ▷ $l$ is now processed
11:                 if $l + w_i \le c/2$ then                 ▷ if the arc is standard
12:                     $\mathcal{A}_s \leftarrow \mathcal{A}_s \cup (l, l + w_i, s)$
13:                     $\mathcal{V} \leftarrow \mathcal{V} \cup \{(l + w_i)\}$
14:                     $M[l + w_i] \leftarrow 1$
15:                 end if
16:                 for $t \in T$ do                          ▷ for each bin type
17:                     if $l + w_i > c_t/2$ and $l \le c_t - (l + w_i)$ then    ▷ reflection in $c_t/2$
18:                         $\mathcal{A}_r(t) \leftarrow \mathcal{A}_r(t) \cup (l, c_t - (l + w_i), t)$
19:                         $\mathcal{V} \leftarrow \mathcal{V} \cup \{(c_t - (l + w_i))\}$
20:                     end if
21:                 end for
22:             end if
23:         end for
24:     end for
25: end for
26: for $t \in T$ do $\mathcal{V} \leftarrow \mathcal{V} \cup \{c_t/2\}$
27: for $i \in \mathcal{V}$ do $\mathcal{A}_s \leftarrow \mathcal{A}_s \cup \{(i, j, s)\colon j = \min(l \in \mathcal{V} : l > i)\}$    ▷ loss arcs
28: for $t \in T$ do $\mathcal{A}_r(t) \leftarrow \mathcal{A}_r(t) \cup (c_t/2, c_t/2, r(t))$    ▷ allow paths that collide in $c_t/2$
29: return $\mathcal{V}, \mathcal{A}_r(1), \dots, \mathcal{A}_r(|T|), \mathcal{A}_s$
```

Appendix I Algorithm for BPPIF

In Algorithm 10, we describe the way in which we build the multigraph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ used in our arc-flow formulations for the BPPIF, in particular, by F_{AF}^{bm} for the bm-BPPIF and by F_{AF}^{fm} for the fm-BPPIF. We recall that $\mathcal{A}_t$ contains all the transposed arcs. Algorithm 10 is similar to the previous Algorithm 7 described for reflect (see Section Appendix G), so we only highlight the differences between them. The arrays M and H now consider the entire range from 0 to c instead of 0 to $c/2$ (steps 3 and 6). Also step 8 is extended by taking into consideration all possible arc tails up to $c - 1$. The standard arcs are created as done for the BPP (and for the VSBPP as well), but the creation of the transposed arcs, performed at steps 16–18, takes into account the fact that if $l + w_j > c$ then the head of the arc is transposed into $(l + w_j) \mod c$.

Algorithm 10 `Create_arcflow_multigraph_BPPIF`

```
 1: Input: $c$: Bin capacity, $w$: Item widths, $d$: Item demands
 2: $\mathcal{A}_s \leftarrow \emptyset$ ; $\mathcal{A}_t \leftarrow \emptyset$ ; $\mathcal{V} \leftarrow \emptyset$
 3: $M[0 \dots c] \leftarrow 0$                    ▷ an array that keeps track of the possible tails
 4: $M[0] \leftarrow 1$                           ▷ node 0 is a possible tail
 5: for $i = 1$ to $m$ do
 6:     $H[0 \dots c] \leftarrow 0$               ▷ an array that keeps track of the tails already processed
 7:     for $j = 1$ to $d_i$ do
 8:         for $l = c - 1$ down to 0 do
 9:             if $H[l] = 0$ and $M[l] = 1$ then   ▷ if $l$ is a possible tail and was not processed yet
10:                 $H[l] = 1$                    ▷ $l$ is now processed
11:                 if $l + w_i \leq c$ then      ▷ if the arc is standard
12:                     $\mathcal{A}_s \leftarrow \mathcal{A}_s \cup (l, l + w_i)$
13:                     $\mathcal{V} \leftarrow \mathcal{V} \cup \{(l + w_i)\}$
14:                     $M[l + w_i] \leftarrow 1$
15:                 else                          ▷ the arc is transposed
16:                     $\mathcal{A}_t \leftarrow \mathcal{A}_t \cup (l, (l + w_i) \mod c)$
17:                     $\mathcal{V} \leftarrow \mathcal{V} \cup \{((l + w_i) \mod c)\}$
18:                     $M[(l + w_i) \mod c] \leftarrow 1$
19:                 end if
20:             end if
21:         end for
22:     end for
23: end for
24: $\mathcal{V} \leftarrow \mathcal{V} \cup \{c\}$
25: for $i \in \mathcal{V}$ do $\mathcal{A}_s \leftarrow \mathcal{A}_s \cup \{(i, j)\colon j = \min(l \in \mathcal{V} : l > i)\}$   ▷ loss arcs
26: return $\mathcal{V}, \mathcal{A}_s, \mathcal{A}_t$
```

Acknowledgements

We gratefully acknowledge support by MIUR-Italy under Grant PRIN 2015. We also thank Gleb Belov for helping us in running his optimization code.

References

Ahuja RK, Magnanti TL, Orlin JB (1993) *Network flows: theory, algorithms, and applications* (Upper Saddle River, NJ, USA: Prentice-Hall).

Alves C, Clautiaux F, Valério de Carvalho J, Rietz J (2016) *Dual-Feasible Functions for Integer Programming and Combinatorial Optimization*. EURO Advanced Tutorials on Operational Research (Springer International Publishing).

Alves C, Valério de Carvalho J (2008) A stabilized branch-and-price-and-cut algorithm for the multiple length cutting stock problem. *Computers & Operations Research* 35:1315–1328.

Arbib C, Marinelli F, Rossi F, Di Iorio F (2002) Cutting and reuse: An application from automobile component manufacturing. *Operations Research* 50:923–934.

Belov G, Scheithauer G (2002) A cutting plane algorithm for the one-dimensional cutting stock problem with multiple stock lengths. *European Journal of Operational Research* 141:274–294.

Belov G, Scheithauer G (2006) A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting. *European Journal of Operational Research* 171:85–106.

Boschetti M, Montaletti L (2010) An exact algorithm for the two-dimensional strip-packing problem. *Operations Research* 58:1774–1791.

Brandão F, Pedroso J (2016) Bin packing and related problems: General arc-flow formulation with graph compression. *Computers & Operations Research* 69:56–67.

Cambazard H, O'Sullivan B (2010) Propagating the bin packing constraint using linear programming. *Principles and Practice of Constraint Programming–CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, 129–136 (Springer-Verlag).

Caprara A, Dell'Amico M, Díaz Díaz J, Iori M, Rizzi R (2015) Friendly bin packing instances without integer round-up property. *Mathematical Programming* 150:5–17.

Casazza M, Ceselli A (2016) Exactly solving packing problems with fragmentation. *Computers & Operations Research* 75:202–213.

Ceselli A, Righini G (2008) An optimization algorithm for the ordered open-end bin-packing problem. *Operations Research* 56:425–436.

Clautiaux F, Hanafi S, Macedo R, Voge ME, Alves C (2017) Iterative aggregation and disaggregation algorithm for pseudo-polynomial network flow models with side constraints. *European Journal of Operational Research* 258:467–477.

Coffman E, Csirik J, Galambos G, Martello S, Vigo D (2013) Bin packing approximation algorithms: Survey and classification. Pardalos P, Du DZ, Graham R, eds., *Handbook of Combinatorial Optimization* (Springer New York).

Correia I, Gouveia L, Saldanha-da-Gama F (2008) Solving the variable size bin packing problem with discretized formulations. *Computers & Operations Research* 35:2103–2113.

Côté JF, Dell'Amico M, Iori M (2014) Combinatorial Benders' cuts for the strip packing problem. *Operations Research* 62:643–661.

Côté JF, Iori M (2016) The meet-in-the-middle principle for cutting and packing problems. Technical Report CIRRELT-2016-28, CIRRELT, Montreal, Canada.

Crainic T, Perboli G, Rei W, Tadei R (2011) Efficient lower bounds and heuristics for the variable cost and size bin packing problem. *Computers & Operations Research* 38:1474–1482.

Dell'Amico M, Díaz-Díaz J, Iori M (2012) The bin packing problem with precedence constraints. *Operations Research* 60:1491–1504.

Delorme M, Iori M, Martello S (2016) Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research* 255:1–20.

Delorme M, Iori M, Martello S (2017a) BPPLIB: A library for bin packing and cutting stock problems. *Optimization Letters* Forthcoming.

Delorme M, Iori M, Martello S (2017b) Logic based Benders' decomposition for orthogonal stock cutting problems. *Computers & Operations Research* 78:290 – 298.

Dyckhoff H (1981) A new linear programming approach to the cutting stock problem. *Operations Research* 29:1092–1104.

Furini F, Malaguti E, Thomopulos D (2016) Modeling two-dimensional guillotine cutting problems via integer programming. *INFORMS Journal on Computing* 28:736–751.

Gilmore P, Gomory R (1961) A linear programming approach to the cutting-stock problem. *Operations Research* 9:849–859.

Gschwind T, Irnich S (2016) Dual inequalities for stabilized column generation revisited. *INFORMS Journal on Computing* 28:175–194.

Hemmelmayr V, Schmid V, Blum C (2012) Variable neighbourhood search for the variable sized bin packing problem. *Computers & Operations Research* 39:1097–1108.

Kartak V, Ripatti A, Scheithauer G, Kurz S (2015) Minimal proper non-IRUP instances of the one-dimensional cutting stock problem. *Discrete Applied Mathematics* 187:120–129.

Macedo R, Alves C, Valério de Carvalho J, Clautiaux F, Hanafi S (2011) Solving the vehicle routing problem with time windows and multiple routes exactly using a pseudo-polynomial model. *European Journal of Operational Research* 214:536–545.

Martello S, Pisinger D, Toth P (1999) Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science* 45:414–424.

Nitsche C, Scheithauer G, Terno J (1999) Tighter relaxations for the cutting stock problem. *European Journal of Operational Research* 112:654–663.

Pessoa A, Uchoa E, Poggi de Aragão M, Rodrigues R (2010) Exact algorithm over an arc-time-indexed formulation for parallel machine scheduling problems. *Mathematical Programming Computation* 2:259–290.

Rao M (1976) On the cutting stock problem. *Journal of the Computer Society of India* 7:35–39.

Sadykov R, Vanderbeck F (2013) Bin packing with conflicts: A generic branch-and-price algorithm. *INFORMS Journal on Computing* 25(2):244–255.

Sadykov R, Vanderbeck F, Pessoa A, Tahiri I, Uchoa E (2016) Primal Heuristics for Branch-and-Price: the assets of diving methods, URL `https://hal.inria.fr/hal-01237204`, working paper or preprint.

Shapiro J (1968) Dynamic programming algorithms for the integer programming problem-I: The integer programming problem viewed as a knapsack type problem. *Operations Research* 16:103–121.

Valério de Carvalho J (1999) Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research* 86:629–659.

Valério de Carvalho J (2002) LP models for bin packing and cutting stock problems. *European Journal of Operational Research* 141:253–273.

Valério de Carvalho J (2005) Using extra dual cuts to accelerate column generation. *INFORMS Journal on Computing* 17:175–182.

Vanderbeck F (1999) Computational study of a column generation algorithm for bin packing and cutting stock problems. *Mathematical Programming* 86:565–594.

Vanderbeck F, Wolsey L (2010) Reformulation and decomposition of integer programs. Jünger M, Liebling T, Naddef D, Nemhauser G, Pulleyblank W, Reinelt G, Rinaldi G, Wolsey L, eds., *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, 431–502 (Berlin, Heidelberg: Springer Berlin Heidelberg).

Wolsey L (1977) Valid inequalities, covering problems and discrete dynamic programs. *Annals of Discrete Mathematics* 1:527–538.