Discrete Optimization 10 (2013) 210–223

Contents lists available at ScienceDirect

Discrete Optimization

journal homepage: www.elsevier.com/locate/disopt

Exact algorithms for the bin packing problem with fragile objects

Manuel A. Alba Martínez [a], François Clautiaux [b], Mauro Dell'Amico [a], Manuel Iori [a,*]

[a] University of Modena and Reggio Emilia, Via Amendola 2, 42122 Reggio Emilia, Italy

[b] Université de Lille 1, LIFL, UMR CNRS 8022, INRIA Lille Nord Europe, Parc de la Haute Borne, 59655 Villeneuve d'Ascq, France

ARTICLE INFO

Article history:
Received 13 June 2012
Received in revised form 6 June 2013
Accepted 15 June 2013
Available online 17 July 2013

Keywords:
Bin packing
Fragile objects
Branch-and-bound
Branch-and-price

ABSTRACT

We are given a set of objects, each characterized by a weight and a fragility, and a large number of uncapacitated bins. Our aim is to find the minimum number of bins needed to pack all objects, in such a way that in each bin the sum of the object weights is less than or equal to the smallest fragility of an object in the bin. The problem is known in the literature as the Bin Packing Problem with Fragile Objects, and appears in the telecommunication field, when one has to assign cellular calls to available channels by ensuring that the total noise in a channel does not exceed the noise acceptance limit of a call.

We propose a branch-and-bound and several branch-and-price algorithms for the exact solution of the problem, and improve their performance by the use of lower bounds and tailored optimization techniques. In addition we also develop algorithms for the optimal solution of the related knapsack problem with fragile objects. We conduct an extensive computational evaluation on the benchmark set of instances, and show that the proposed algorithms perform very well.

1. Introduction

In the *Bin Packing Problem with Fragile Objects* (BPPFO), we are given a set $N = \{1, 2, \ldots, n\}$ of objects and m bins. Each object j is characterized by a weight w_j and a fragility f_j ($j \in N$), whereas the bins are uncapacitated. The BPPFO is to pack all objects in the minimum number of bins, guaranteeing that in each bin the sum of the object weights does not exceed the smallest fragility of an object. Stated more formally, let us define $S(i)$ the set of objects assigned to a bin i ($i = 1, 2, \ldots, m$). A solution to the BPPFO is feasible if each object is assigned exactly to one bin, and for any bin i

$$\sum_{j \in S(i)} w_j \leq \min_{j \in S(i)} \{f_j\} \tag{1}$$

holds. In the following without loss of generality we suppose that $w_j \leq f_j$, for $j \in N$, and that m is large enough to guarantee that a feasible solution exists (e.g., $m = n$ is always enough). We also suppose that objects are sorted according to

$$f_j \leq f_{j+1} \quad j = 1, 2, \ldots, n-1. \tag{2}$$

A simple example with 6 objects is depicted in Fig. 1(a): object 1 has weight 2 and fragility 5, object 2 has weight 5 and fragility 6, and so on. An optimal solution using 4 bins is shown in Fig. 1(c) (disregard for the moment Fig. 1(b)).

* Corresponding author. Tel.: +39 0522 522 653.
E-mail addresses: manuel.alba@unimore.it (M.A. Alba Martínez), francois.clautiaux@univ-lille1.fr (F. Clautiaux), mauro.dellamico@unimore.it (M. Dell'Amico), manuel.iori@unimore.it, manuel.iori@gmail.com (M. Iori).

1572-5286/$ – see front matter
http://dx.doi.org/10.1016/j.disopt.2013.06.001

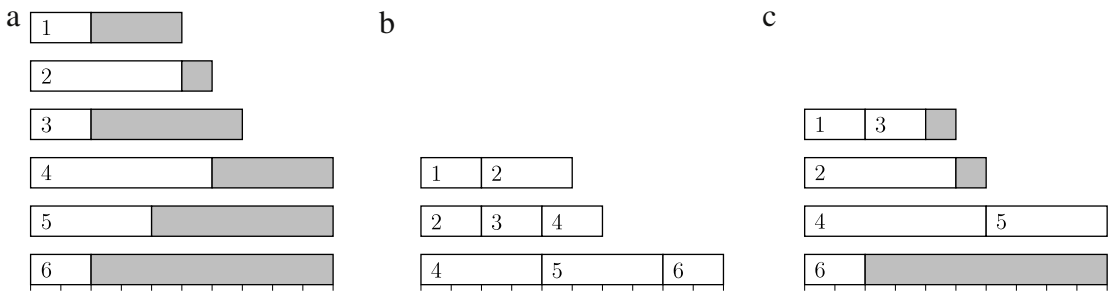

Fig. 1. (a) A BPPFO instance; (b) the relaxation obtained by applying L_2; (c) an optimal solution.

The BPPFO is important in telecommunications, where it models the allocation of cellular calls to frequency channels. In *Code Division Multiple Access* (CDMA) systems, a network is divided into cells, and each cell is equipped with an antenna having a limited number of frequency channels. Any time a user makes a call, the CDMA system assigns the call to a frequency channel.

Each call is characterized by a certain noise that it produces, and by a certain tolerance with respect to the total noise in the channel. These parameters have to be taken into account when performing the assignment. Indeed many calls may be assigned to the same channel, because the bandwidth capacity of a channel is much larger than the bandwidth requirement of any call, but such assignments may produce interferences among calls sharing the same channel because the sum of the noises may be too high to guarantee a good communication. For such a reason, it is required that the total noise in a frequency channel does not exceed the tolerance of any call assigned to the channel.

In the seminal paper by Bansal et al. [1] the problem of assigning calls in a CDMA system is modeled as a BPPFO by associating frequency channels to bins, and cellular calls to objects having weights equal to the call noises and fragilities equal to the call tolerances. By solving the BPPFO one determines a feasible assignment of the cellular calls to the minimum number of frequency channels.

The BPPFO is a difficult problem. Consider indeed the *Bin Packing Problem* (BPP): we are given n objects of weight w_j ($j \in N$) and a large number of bins of capacity C, with the aim of packing the objects in the minimum number of bins, and ensuring that the sum of the objects' weights does not exceed the bin capacity in any bin. One can see that the BPP is a particular BPPFO where $f_j = C$ for all $j \in N$. Since the BPP is strongly $\mathcal{NP}$-hard, the same holds for the BPPFO. It is worth mentioning that the BPPFO is also very difficult in practice: there are benchmark instances with just 50 objects that are unsolved to proven optimality, see Clautiaux et al. [2]. Note that there is a huge literature on the BPP that we do not survey in this work. The interested reader is referred to, e.g., Valério de Carvalho et al. [3], Clautiaux et al. [4] and Coffman et al. [5]. We just mention the fact that the best performing algorithms for the BPP are based on branch-and-price, see, e.g., Alves and Valério de Carvalho [6] and Vanderbeck [7], and this is indeed the main technique that we use in this work to tackle the BPPFO.

The aim of this paper is indeed to present exact algorithms for the BPPFO. In particular we make use of new and existing lower and upper bounding techniques and embed them into ad hoc enumeration schemes. We produce several branch-and-price algorithms that improve in a consistent way the number of proven optimal solutions for a benchmark set of instances. During the process we also develop efficient algorithms for the knapsack problem with fragile objects, that is the slave problem to be solved when doing column generation for the BPPFO.

The remainder of the paper is organized as follows. To have a self-contained paper, in Section 2 we survey the main results by the literature. In Section 3 we address the problem of solving the knapsack problem with fragile objects, a subproblem that naturally arises when addressing the BPPFO through column generation. In Section 4 we embed the techniques developed in the previous section into branch-and-price algorithms that make use of binary branching schemes. In Section 5 we present two algorithms, a combinatorial branch-and-bound and a branch-and-price, that make use of a non-binary branching rule. Extensive computational results are given in Section 6 and conclusions are drawn in Section 7.

2. Prior work on the problem

Chan et al. [8] studied an on-line version of the BPPFO in which information on an object is available only after the previous object has been allocated to a bin. They solved the problem through approximation schemes and presented a simple lower bound obtained by computing

$$L_1 = \left\lceil \sum_{j=1}^{n} \frac{w_j}{f_j} \right\rceil. \tag{3}$$

L_1 can be quickly computed in $O(n)$, but is usually bad in practice. A better bound, called L_2, was proposed by Bansal et al. [9], and is based on the solution of the following combinatorial problem:

the *Fractional Bin Packing Problem with Fragile Objects* (FBPPFO) is the relaxation of the BPPFO in which an object is allowed to split up arbitrarily into several pieces, and these pieces can be packed into different bins. Each piece takes the fragility of the object from which it is cut.

Recall that objects are sorted by non-decreasing fragilities, i.e., according to (2). Bansal et al. [9] prove that the FBPPFO can be solved exactly through the following polynomial algorithm: open a first bin of capacity f_1 and pack object 1 in it; for any successive object j, if j fits in the current bin then pack j in it, otherwise pack in the current bin only the fraction of j that fits, open a new bin of capacity f_j and pack the residual fraction of j in the new bin. In the following we will call L_2 both this algorithm and the value it outputs. The pseudo-code of L_2 is given in Algorithm 1.

input : n objects sorted by non-decreasing fragility

$L_2 := 1$;
$f_{res} := f_1 - w_1$;
for $j := 2$ **to** n **do**
  if $w_j \leq f_{res}$ **then**
    $f_{res} := f_{res} - w_j$;
  else
    $L_2 := L_2 + 1$; // open a new bin of capacity f_j
    $w_{res} := w_j - f_{res}$;
    $f_{res} := f_j - w_{res}$;
  end
end
return L_2;

Algorithm 1: Computation of the optimal solution of the FBPPFO with lower bound L_2.

Fig. 1(b) reports the application of L_2 to the example shown in Fig. 1(a). The algorithm runs in $O(n)$ time once the objects are sorted accordingly to (2). It dominates L_1 and is quite effective in practice.

For what concerns mathematical models, Clautiaux et al. [2] modeled the BPPFO as an *integer linear program* (ILP) by using a binary variable y_i taking value 1 if object i is the object with smallest fragility in the bin in which it is packed, 0 otherwise ($i \in N$), and another binary variable x_{ji} taking value 1 if object j is assigned to the bin having object i as object with smallest fragility, 0 otherwise ($i = 1, 2, \ldots, n, j = i + 1, i + 2, \ldots, n$). The resulting ILP is:

$$\min \quad \sum_{i=1}^{n} y_i \tag{4}$$

$$y_j + \sum_{i=1}^{j-1} x_{ji} = 1 \quad j = 1, 2, \ldots, n \tag{5}$$

$$\sum_{j=i+1}^{n} w_j x_{ji} \leq (f_i - w_i) y_i \quad i = 1, 2, \ldots, n \tag{6}$$

$$x_{ji} \leq y_i \quad i = 1, 2, \ldots, n, j = i + 1, i + 2, \ldots, n \tag{7}$$

$$y_i \in \{0, 1\} \quad i = 1, 2, \ldots, n \tag{8}$$

$$x_{ji} \in \{0, 1\} \quad i = 1, 2, \ldots, n, j = i + 1, i + 2, \ldots, n. \tag{9}$$

The objective function (4) imposes the minimization of the number of bins. Constraints (5) require each object to be packed exactly once, constraints (6) model the fragility restriction and constraints (7) make the model linear relaxation tighter.

Clautiaux et al. [2] also proposed an ILP based on the classical set covering formulation for the BPP. They define a pattern, say p, as a set of objects that satisfies Eq. (1), and P as the set of all patterns. To describe a pattern p they use a vector $(a_{1p}, a_{2p}, \ldots, a_{np})$, where a_{jp} takes value 1 if object j is in the pattern, 0 otherwise, for $p \in P$. Then they introduce a binary variable z_p taking value 1 if pattern p is used, 0 otherwise, for $p \in P$, and model the BPPFO as:

$$\min \quad \sum_{p \in P} z_p \tag{10}$$

$$\sum_{p \in P} a_{jp} z_p \geq 1 \quad j = 1, 2, \ldots, n \tag{11}$$

$$z_p \in \{0, 1\} \quad \forall p \in P. \tag{12}$$

The objective function (10) requires the minimization of the number of bins, and constraints (11) impose that each object j is packed in at least one bin.

The size of P is exponential in the number of objects, so even solving the linear relaxation of (10)–(12) is difficult. Clautiaux et al. [2] tackled this problem by means of a column generation algorithm: they relaxed constraints (12) to linearity and initialized the model with a subset of patterns, then they iteratively added patterns with negative reduced cost, if any, until an optimal (linear) solution was found. The slave problem that needs to be solved to determine if a negative cost pattern exists is a particular knapsack, that was solved with an ILP model (see Section 3.1 below). The model was enriched with dual cuts, see also Valério de Carvalho [10] and Clautiaux et al. [11]. Note that no branching scheme was used to attempt to find an integer solution.

In terms of upper bounds, Bansal et al. [1] presented a 2-approximation greedy heuristic. Clautiaux et al. [2] proposed a large set of greedy heuristics and a *Variable Neighborhood Search* (VNS) that obtained good results on the benchmark set of instances.

Summarizing the results in the literature, the only exact algorithm is the one based on the solution of the ILP model (4)–(9). In this paper we intend to miss this gap by developing new exact algorithms, using the most successful techniques from the literature and proposing new ones.

3. Solution of the knapsack problem with fragile objects

We define two combinatorial problems:

- in the well-known 0–1 *Knapsack Problem* (KP01), n objects with profit p_j and weight w_j ($j \in N$) have to be packed into a single bin of capacity C, with the aim of determining a subset of objects whose total profit is largest and whose total weight does not exceed C;
- in the 0–1 *Knapsack Problem with Fragile Objects* (KP01FO), n objects with profit p_j, weight w_j and fragility f_j ($j \in N$) have to be packed into a single uncapacitated bin, with the aim of determining a subset of objects whose total profit is largest and whose total weight does not exceed the fragility of any object in the bin (according to (1)).

The KP01 is the slave problem that appears in the classical column generation algorithm for the BPP. The KP01FO appears instead in the column generation for the BPPFO (as will be discussed in detail in Section 4). Before introducing our branch-and-price algorithms, we describe here three algorithms to exactly solve the KP01FO.

3.1. An ILP model

The KP01FO can be modeled (see Clautiaux et al. [2]) as an ILP by defining a binary variable β_j taking value 1 if object j is the object with the smallest fragility in the bin, 0 otherwise, and a binary variable α_j taking value 1 if object j is packed in the bin but is not the object with smallest fragility, 0 otherwise, for $j \in N$. The KP01FO is then:

$$\max \quad \sum_{j=1}^{n} p_j(\alpha_j + \beta_j) \tag{13}$$

$$\sum_{j=1}^{n} \beta_j = 1 \tag{14}$$

$$\sum_{j=1}^{n} \beta_j(f_j - w_j) - \sum_{j=1}^{n} w_j\alpha_j \geq 0 \tag{15}$$

$$\alpha_k + \sum_{j=k}^{n} \beta_j \leq 1 \quad k = 1, 2, \ldots, n \tag{16}$$

$$\alpha_j \in \{0, 1\} \quad j = 1, 2, \ldots, n \tag{17}$$

$$\beta_j \in \{0, 1\} \quad j = 1, 2, \ldots, n. \tag{18}$$

Constraints (14) impose the existence of just one object with smallest fragility inside the bin, constraints (15) model the fragility restriction, and constraints (16) force the relation between the two families of variables (recall that objects are sorted according to (2)).

3.2. A KP01-based approach

Let us call "witness" the object having smallest fragility in the bin, i.e., the object j with $\beta_j = 1$ in the optimal solution of the ILP model (13)–(18). Each object in a KP01FO instance is indeed a candidate to be the witness. We may attempt one at a time all objects to be the witness, evaluate the total profit for each such attempt and then keep the best total profit as the optimal solution to the original KP01FO instance.

The total profit obtainable when a given object j is the witness may be computed by solving a KP01 instance in which the bin capacity is fixed to f_j, object j is forced in the bin and objects whose fragility is smaller than f_j are removed from the instance. In practice, we solve a KP01 instance with $C = f_j - w_j$ and objects set $\{j+1, j+2, \ldots, n\}$, and then add profit p_j to the solution value obtained.

Our algorithm, denoted *KP01-based* in the following, performs n attempts starting with object 1, then object 2, and so on. At each attempt j, it solves the KP01 instance just defined and uses the resulting solution to possibly improve the best KP01FO solution obtained so far. Some simple techniques are used to speed up the process. Namely, at a given attempt j:

- we further reduce the current set of objects by removing all objects k having $w_k > f_j - w_j$;
- let us denote z the best solution obtained in the attempts already evaluated. Before invoking the exact KP01 algorithm we compute an upper bound, say u, on the KP01 solution, and if $u \leq z$ then we skip the attempt.

In our implementation the value of u was computed with the upper bound procedure U_2 by Martello and Toth [12], and the KP01 was solved with the routine `combo` by Martello et al. [13] (available at http://www.diku.dk/~pisinger/codes.html).

3.3. Dynamic programming

It is well known that the KP01 can be solved through a *dynamic programming* (DP) algorithm that uses a status $\psi(j, c)$ to store the optimal solution of a sub-instance given by the first j objects and a bin of capacity c. The optimal KP01 solution is given by the status $\psi(n, C)$, computed through the recursion:

$$\psi(j,c) = \begin{cases} \psi(j-1,c) & \text{if } w_j > c \\ \max(\psi(j-1,c), \psi(j-1,c-w_j)+p_j) & \text{if } w_j \leq c \end{cases} \quad c = 0,\ldots,C;\ j = 1,\ldots,n, \tag{19}$$

with the initialization $\psi(0, c) = 0$ for $c = 0, \ldots, C$. This recursion performs one phase for each object, and for each phase considers all the possible bin sizes, so the overall computational complexity is $O(nC)$.

The KP01FO can be solved through DP by using the method of Section 3.2, that consists of solving at most n KP01. For each object j selected as the witness, we solve the corresponding KP01 setting the bin capacity to $f_j - w_j$ and reducing the subset of objects accordingly. The KP01 solution of maximum value is the optimal KP01FO solution. Thus the method has an overall time complexity $O(n^2 f_{\max})$, with $f_{\max} = \max_{j \in N}\{f_j\}$.

A more efficient method can be obtained sorting the objects by *non-increasing* fragility. In this case a unique execution of the DP algorithm solves the KP01FO in $O(nf_{\max})$, as stated by the following:

Proposition 1. *Let the objects of a* KP01FO *instance be sorted by non-increasing fragility and apply recursion (19). The optimal solution of* KP01FO *is given by the state*

$$\psi(k, f_k) = \max_{j \in N}\{\psi(j, f_j)\}. \tag{20}$$

Proof. We first observe that any state $\psi(j, c)$ with $c < f_j$ is dominated by $\psi(j, f_j)$. The states $\psi(j, c)$ with $c > f_j$ are not admissible for the KP01FO if object j is chosen. If instead j is not chosen, then $\psi(j, c)$ is dominated by $\psi(j-1, c)$. It follows that the only state of phase j to be considered for the optimal solution of KP01FO is the one with $c = f_j$.

To complete the proof it is enough to show that a non-admissible state of phase j cannot be used to generate admissible states in the next phases. The recursion (19) uses $\psi(j, c)$ to generate states with bin size not smaller than c. Since the objects are ordered so that $f_j \geq f_{j+1}$, a state of phase $j+1$ generated by $\psi(j, c)$ is always not admissible if object $j+1$ is selected. If otherwise $j+1$ is not selected the state of phase $j+1$ is dominated by $\psi(j, c)$, which concludes the proof. □

A direct consequence of Proposition 1 is that we can avoid generating the states $\psi(j, c)$ with $c > f_j$ and the states $\psi(k, f_j)$ with $k > j$, thus accelerating the computation of the optimal solution.

4. Binary branching schemes

In this section we present two *Branch-and-Price* (B&P) algorithms for the solution of the set covering formulation (10)–(12) of Section 2. Since the formulation has an exponentially large number of columns, the classical column generation approach, originally described for the BPP by Gilmore and Gomory [14,15], is used. The algorithms we propose solve at each node of an enumeration tree the continuous relaxation of the formulation through column generation by iteratively invoking the algorithms of Section 3, then possibly perform branching to obtain an integer solution.

In more detail, we solve the continuous relaxation of (10)–(12) obtained by substituting (12) with $z_p \geq 0$ for all $p \in P$. We initialize the model with a subset of patterns. The resulting problem is called the *restricted master*. Then we associate dual variables π_j to constraints (11), and solve a KP01FO with objects profits π_j and weights w_j. This is called the *slave* problem. If the value of the solution of the slave is larger than 1, then the set of objects in the solution corresponds to a pattern of the set covering formulation having a negative reduced cost. In this case the pattern (i.e., the column) is added to the restricted master and the process is reiterated. Otherwise, the optimal solution of the continuous relaxation has been found and a

valid lower bound has been obtained. Note that we can obtain a valid lower bound also during the iterations of the column generation, and not only at the end. Let us denote by z_{lp} the current solution value of the master and by z_{kp} the current solution value of the slave, then a valid lower bound is $\lceil z_{lp}/z_{kp} \rceil$.

This procedure is applied at each node of an enumeration tree. If the produced lower bound is not smaller than that of the incumbent solution, then the node is fathomed, otherwise branching is performed. The algorithms described in this section perform a binary branching by selecting a fractional component of the current continuous relaxation, if any, and setting it first to 1 and then to 0. The tree created by the adopted branching rule is explored in depth first fashion. Each node of the tree "inherits" from its parent node the columns of the restricted master and the branching constraints.

Branching is a very critical point in B&P algorithms, because the additional branching constraint must be taken into consideration both in the restricted master (usually by removing columns) and in the slave (that must not recreate columns that have just been removed). There are some branching schemes that do not modify the structure of the slave problem, as the one presented in Vanderbeck [7]. However, this is not the case for the binary schemes presented here, that typically affect the slave problem and may have a strong computational impact. In the remainder of the section we present two binary branching schemes and several algorithms for solving the resulting slave problems.

4.1. Branching scheme 1

The first branching scheme we implemented (B&P-1) is based upon the idea of branching on patterns having a fractional value. Suppose a fractional solution z_p^*, $p \in P$, is found at a given node. We select the pattern, say q, for which the value of z_q^* is closest to 0.5, and branch first to 1 and then to 0. We define by

$$Q = \{j \in N : a_{jq} = 1\}$$

the set of objects included in pattern q, $q \in P$.

When branching to $z_q = 1$, we simply remove from the instance all objects in Q. In the column generation phase, the slave problem is still a pure KP01FO, but with a reduced subset of objects, hence can be solved with any algorithm of Section 3.

When branching to $z_q = 0$, to generate new valid columns we need to include in the slave problem a constraint imposing that not all the objects contained in Q are inserted together in a newly generated pattern. This changes the nature of the subproblem, which becomes strongly $\mathcal{NP}$-hard and cannot be solved with either the KP01-based approach or the DP algorithm (both having pseudo-polynomial complexity). To see that the new problem is strongly $\mathcal{NP}$-hard we restrict the attention to the KP01, instead of the more general KP01FO. Suppose that all sets Q corresponding to the added branching constraints contain exactly two objects. These pairs of objects are *in conflict*, one with the other, and cannot be packed together in a bin. The resulting problem is known as the *knapsack problem with conflicts*, see, e.g., Fernandes Muritiba et al. [16], and was proved to be strongly $\mathcal{NP}$-hard in Pferschy and Schauer [17].

In our approach we solve the resulting KP01FO through the ILP of Section 3.1, and model the new restriction by adding the constraint

$$\sum_{j \in Q} (\alpha_j + \beta_j) \leq |Q| - 1. \tag{21}$$

The number of branches to 0 can be quite high, and for each of them we need to add a constraint (21). Thus the resulting ILP model may be quite large at nodes that are deep in the tree. Summarizing, we can solve the slave with any algorithm as long as we branch to $z_q = 1$, but we need to use the ILP at all nodes derived from a branching to $z_q = 0$.

4.2. Branching scheme 2

The second branching scheme we implemented (B&P-2) is based on the variables y_j and x_{ji} introduced in model (4)–(9) of Section 2. Recall the witness object is the object having the smallest fragility in the bin. We define:

$$P(j) = \{\text{set of patterns in which object } j \text{ is the witness}\},$$

for $j \in N$. Once we have the current continuous solution z_p^* to (10)–(12), we compute the values of the variables y_j^* and x_{ji}^* using:

$$y_j^* = \sum_{p \in P(j)} z_p^* \quad j = 1, 2, \ldots, n, \tag{22}$$

$$x_{ji} = \sum_{p \in P(i)} a_{jp} z_p^* \quad i = 1, 2, \ldots, n, j = i+1, i+2, \ldots, n. \tag{23}$$

If all y_j^* and x_{ji}^* variables take an integer value, then we found a feasible solution to the instance, hence, after possibly improving the incumbent solution, we fathom the node. If instead some y_j^* values are fractional, then we take the y_j^* having the closest value to 1/2, and create two branches that are equivalent to set y_j^* first to 1 and then to 0.

Table 1
Imposing branching directions in master and slave problems in B&P-2.

Branching direction	Columns removed from (10)–(12)	Slave solution		
		ILP	KP01-based	DP
$y_j = 1$	$\{p \in P \setminus P(j) : a_{jp} = 1\}$	add $\alpha_j = 0$	KP01-based-$y1$	DP-$y1$
$y_j = 0$	$\{p \in P(j)\}$	add $\beta_j = 0$	KP01-based-$y0$	DP-$y0$
$x_{ji} = 1$	$P(j) \cup \{p \in P \setminus P(i) : a_{jp} = 0\}$	composite object	composite object	composite object
$x_{ji} = 0$	$\{p \in P(i) : a_{jp} = 1\}$	add $\beta_i + \alpha_j \leq 1$	KP01-based-$x0$	n.a.

We note that in this enumeration tree it is possible to find a node in which $y_j^* > 1$ for a certain object j. Suppose for example that $n = 4$ and the current set of patterns contain patterns $(1, 1, 1, 0)$, $(1, 1, 0, 1)$ and $(1, 0, 1, 1)$. Then the optimal solution to (10)–(12) consists of selecting each pattern with value $1/2$, hence obtaining $y_1^* = 3/2$. In this case branching to $y_1^* = 1$ would produce no effect and the algorithm would enter a loop. To overcome this issue we replaced "$\geq$" with "$=$" in (11), i.e., we work with a set partitioning formulation instead of a set covering one. Note that this typically increases the number of columns to be generated, since one cannot restrict the computation to maximal patterns but must also consider all proper subsets. However, as shown in Section 6, the number of columns remained reasonable in our computational experiments.

When branching to $y_j = 1$, we remove from (10)–(12) all columns containing object j together with one or more objects of smaller fragility. Note indeed that $y_j = 1$ implies $x_{ji} = 0$ for all i because of (5). The columns that have been removed from the master also need to be excluded from the solutions of the slave problem. This is achieved as follows:

1. when using the ILP model of Section 3.1, we add the constraint $\alpha_j = 0$ to (13)–(18); hence j is allowed to enter the knapsack only as the witness;
2. when using the KP01-based approach of Section 3.2, we remove j from all calls to KP01 having as witness an object i with index $i < j$. This implies that we are not generating any column leading to $x_{ji} = 1$. The calls to KP01 having witness i with $i \geq j$ are performed as usual. We call the resulting algorithm KP01-based-$y1$;
3. when using the DP of Section 3.3, we set $\psi(j, c) = \psi(j-1, c)$ in Eq. (19) whenever $c < f_j$. At the end of the DP process, if the optimal solution corresponds to the state $\psi(j, f_j)$, then it contains object j as the witness (note indeed that $\psi(j, f_j)$ is dominated by $\psi(j-1, f_{j-1})$ if object j is not selected). If instead the optimal solution corresponds to other states, then it does not contain object j because of the modification induced in (19). It follows that j can be used, but only as the witness. The resulting algorithm, that we call DP-$y1$, has the same complexity of DP.

When branching to $y_j = 0$, we remove from (10)–(12) all columns $p \in P(j)$. To exclude these columns from the solutions of the slave problem, we operate as follows:

1. when using the ILP model, we add the constraint $\beta_j = 0$;
2. when using the KP01-based approach, we remove the call to the KP01 having j as witness object. All other calls are performed as usual (note that j can still appear in columns having an object i of smaller fragility as witness, and that would lead to $x_{ji} = 1$). We call the resulting algorithm KP01-based-$y0$;
3. when using the DP, we need to exclude the states having j as object of smallest fragility from the list of states that are candidate to be the optimal solution. This is achieved by removing index j from the max computation in Eq. (20). The resulting algorithm, that we call DP-$y0$, has the same complexity of DP.

If no variable y_j^* having a fractional value exists, then we consider the variable x_{ji}^* having the closest value to $1/2$, if any, and create two branches that are equivalent to set x_{ji}^* first to 1 and then to 0.

When branching to $x_{ji} = 1$, we remove from (10)–(12) all columns in $P(i)$ not containing object j, and all columns in $P(j)$. We then modify the current set of objects by creating a new *composite object*, say k, having weight $w_k = w_i + w_j$, fragility $f_k = f_i$ and profit $p_k = p_i + p_j$. We then replace in the current set of objects j and i with k. The resulting slave problem is a pure KP01FO on a reduced set of objects, hence can be solved with any algorithm of Section 3.

When branching to $x_{ji} = 0$, we remove from (10)–(12) all columns in $P(i)$ that contain object j. To exclude these columns from the solutions of the slave problem, we use the following ideas:

1. when using the ILP model, we add the constraint $\alpha_j + \beta_i \leq 1$;
2. when using the KP01-based approach, when performing the attempt in which object i is selected as the witness, we remove object j from the set of objects in the KP01 instance. All other attempts are performed as usual (hence j can still appear in a column, but not in one having i as witness). We call the resulting algorithm KP01-based-$x0$.

Note that we cannot use the DP after a branch to $x_{ji} = 0$, because, when computing the states associated to i we would need to know what was done in the previous states associated to j, hence a state does not depend only on its immediate predecessor and the dynamic recursion does not work anymore.

The summary of the columns removed from the master and the algorithms used to solve the slave in B&P-2 is given in Table 1. In practice we can use any algorithm as long as we branch to $y_j = 1$, $y_j = 0$ and $x_{ji} = 1$, but we cannot use the DP at all nodes derived from a branching to $x_{ji} = 0$.

4.3. Embedding L_2 into branching scheme 2

It is computationally useful to have a quick lower bound to possibly fathom a node also when some variables have been fixed by branching. We do this by solving the FBPPFO (see Section 2) at a given node of the B&P-2 tree by a modified version of L_2.

In B&P-2 we first fix to integrality the values of fractional y_j variables, while we fix the values of fractional x_{ji} variables only when all y_j have integer value. Let us consider the first levels of the decision tree, i.e., the ones in which we only deal with branchings involving the y_j variables. Given a decision node let $Y1$ denote the set of objects selected as a witness (i.e., $y_j = 1, \forall j \in Y1$), let $Y0$ denote the set of objects fixed not to be a witness (i.e., $y_j = 0, \forall j \in Y0$), and let YF denote the "free" objects (i.e., $YF = N \setminus (Y1 \cup Y0)$). Our aim is to design an algorithm to solve the FBPPFO with y_j variables fixed to 1 or 0 accordingly to sets $Y1$ and $Y0$. We will call this problem the *FBPPFO with imposed and forbidden witness sets* $Y1$ *and* $Y0$. The following lemma is the key element we use to construct our algorithm.

Lemma 1. *Given a FBPPFO instance and a binary vector* y *of* n *elements, consider the system of inequalities*

$$\sum_{j=1}^{k} f_j y_j \geq \sum_{j=1}^{k} w_j \quad k = 1, 2, \ldots, n. \tag{24}$$

The FBPFFO has a feasible solution with witness set $Y1 = \{j \in N : y_j = 1\}$ *if and only if* (24) *holds.*

Proof. The vector y is the first set of variable in the ILP model (4)–(9), i.e., $y_j = 1$ if object j is chosen as a witness, and $y_j = 0$ otherwise.

It is not difficult to see that the condition is necessary. Indeed, due to ordering (2), the pieces of each object k can be packed only in bins having as witness one of the first k objects. If the y vector does not satisfy (24), then there is not enough capacity in these bins to pack the objects $1, 2, \ldots, k$.

To prove that the condition is sufficient, it is enough to show how one can build a feasible solution from a y vector satisfying (24). We start by observing that the first inequality implies $y_1 = 1$, so we pack object 1 in the first bin. For each object $i > 1$ such that $y_i = 1$ we open a bin of capacity f_i and pack i in it. Then we take each unpacked object j, in turn, and pack it fractionally in the open bins with a witness of index smaller than j. This is always possible since condition (24) holds. □

The proof of the above lemma contains a constructive algorithm that is strongly related to Algorithm 1. The difference between the two methods is in the choice of the witnesses. Algorithm 1 defines the witnesses on-the-fly, by minimizing the number of witnesses (or bins) selected. On the contrary Lemma 1 takes some witnesses as an input. In practice Algorithm 1 solves the problem

$$(F0) : \min \left\{ \sum_{j \in N} y_j : (24); y_j \in \{0, 1\}, j \in N \right\}$$

whereas in the context of B&P-2 we solve the more general problem

$$(F1) : \min \left\{ \sum_{j \in N} y_j : (24); y_j = 0, j \in Y0; y_j = 1, j \in Y1; y_j \in \{0, 1\}, j \in YF \right\}.$$

We show that the main difference between the two problems is due to the presence of set $Y0$, which imposes that some objects cannot be used as witness.

The optimal solution to problem $(F0)$ is obtained by opening a new bin associated with witness k when the current object k cannot be entirely packed in the previously opened bins. We know that we can pack k either opening some bins associated with a previous object (i.e., selecting as witness one or more objects $i < k$) or opening a new bin with witness k. In problem $(F0)$ the latter choice leads to the optimal solution since $f_k \geq w_k$ and $f_k \geq f_i$ for all $i < k$.

Now consider a restriction of $(F1)$ where set $Y0$ is empty. It is not difficult to see that we can find the optimal solution of this problem by using a slight modification of Algorithm 1, that just increases the residual capacity f_{res} by $f_j - w_j$ when it encounters an object $j \in Y1$ (which corresponds to open a new bin of capacity f_j and to pack in it object j).

We are now ready to consider the complete problem $(F1)$. In this case, when an object $k \in Y0$ cannot be entirely packed in the previously opened bins, it cannot be packed in a new bin having itself as witness. So we need to open one or more bins using as witness objects with index smaller than k. The optimal choice for these new witnesses is given by the set of objects of maximum fragility taken from the objects of YF that have not yet been used as witnesses. We have thus proved the following:

Proposition 2. *The problem FBPPFO with imposed and forbidden witness sets* $Y1$ *and* $Y0$ *can be solved by Algorithm* 2.

At each iteration of the "for" loop of Algorithm 2, we manage separately objects of $Y1$ from those of $YF \cup Y0$. If $k \in Y1$, then a new bin is opened with witness object k. Otherwise the "while" loop opens new bins corresponding to objects of YF

with index smaller or equal to k and not yet used as witness. If $k \in YF$ we select it as a witness, as done by Algorithm 1 (recall that $f_k \geq w_k$). If otherwise $k \in Y0$, then we consider the objects by decreasing fragility and we select the minimum number of objects that allow to pack k.

The algorithm returns "infeasible" whenever system (24) corresponding to the FBPPFO with imposed branching sets $Y1$ and $Y0$ has no solution. Otherwise it returns the minimum number of bins L_2^{01} to be opened to solve the problem. The value of L_2^{01} is used to possibly fathom the node in B&P-2 before invoking the more time-consuming column generation lower bound.

Finally observe that Algorithm 2 does not provide the complete FBPPFO solution, but only the witnesses set. One can obtain such a solution with the constructive algorithm described in the proof of Lemma 1.

input : $Y1 = \{j \in N : y_j = 1\}$, $Y0 = \{j \in N : y_0 = 1\}$ and $YF = N \setminus (Y1 \cup Y0)$;
for $i := 1$ **to** n **do** $y_i := 0$;
$L_2^{01} := 0$; $r := 0$;
for $k := 1$ **to** n **do**
  if $k \in Y1$ **then**
    $r := r + f_k - w_k$; $y_k := 1$; $L_2^{01} := L_2^{01} + 1$;
  else
    while $r < w_k$ **do**
      // open new bins to accommodate w_k
      $i := \max\{h : h \leq k, h \in YF, y_h = 0\}$;
      if *there is no such* i **then return** *infeasible*; // system (24) has no solution
      $r := r + f_i$; $y_i := 1$; $L_2^{01} := L_2^{01} + 1$;
    end
    $r := r - w_k$;
  end
end
return L_2^{01}

Algorithm 2: Computation of the optimal solution value of the FBPPFO with imposed and forbidden witness sets.

5. Non-binary branching schemes

In this section we present some algorithms making use of a non-binary branching scheme in which at each level an object is packed in a bin. The main difference with respect to the algorithm presented in Section 4 is that the new scheme is not based on the fractional components of the current (relaxed) solution, but uses a pre-specified objects ordering. The scheme is outlined in detail in Section 5.1, together with a combinatorial branch-and-bound. Then it is used in Section 5.2 to obtain a new branch-and-price algorithm.

5.1. A combinatorial branch-and-bound algorithm

The *Branch-and-Bound* (B&B) algorithm we implemented attempts to pack one object at a time according to the initial object ordering (see (2)). At level j of the enumeration tree, the B&B creates a node for any bin that has already been opened and packs object j in such a bin, provided that this does not violate the fragility restriction. The B&B also creates an additional node by opening a new bin of fragility f_j (called, for short, bin j in the following) and packing object j in it. In practice, at level 1 object 1 is packed in bin 1, at level 2 object 2 is packed in bin 1 or in bin 2, and so on.

At each node we need to solve a BPPFO where the first k objects have already been packed. Let us define the following problems with *sequential packing decisions*:

> the BPPFO-s (resp. FBPPFO-s) is the version of the BPPFO (resp. FBPPFO) in which a set $N_1 = \{1, 2, \ldots, k\}$ of objects has already been packed by using z_1 bins, and a set $N_2 = \{k + 1, k + 2, \ldots, n\}$ still has to be packed.

We use $f_{res}(i)$ to denote the residual capacity in bin i ($i = 1, 2, \ldots, z_1$). Whenever a new object is added to N_1 and packed, say in bin i, we use two simple rules to tighten the values of $f_{res}(i)$. Namely:

- if among the objects of N_2 there is no object with weight smaller or equal to $f_{res}(i)$, then we close the bin by setting $f_{res}(i) = 0$;
- if only one object $j \in N_2$ fits the residual capacity of bin i, then we pack j in i and close the bin by setting $f_{res}(i) = 0$.

At any node of the enumeration tree, we use as lower bound the value of the optimal solution of the FBPPFO-s, that we obtain with L_2^{01} (see Algorithm 2). This is done as follows.

Recall L_2^{01} receives in input three distinct objects sets: $Y1$ denotes the set of objects selected as a witness, $Y0$ denotes the set of objects fixed not to be a witness, and YF denotes the set of objects for which no branching decision has been taken.

Table 2
Impact of the slave solution algorithms on the three branching schemes.

	Initialization L_2 + VNS	B&P-1			B&P-2			B&P-3		
		ILP	KP01-b.	DP	ILP	KP01-b.	DP	ILP	KP01-b.	DP
Gap	2.27%	1.73%	1.09%	1.05%	1.61%	1.05%	0.91%	1.53%	0.82%	0.78%
Opt	*312*	*373*	*475*	***479***	*387*	*464*	***473***	*390*	*487*	***495***
Sec	82.13	445.65	319.30	314.64	434.66	352.00	313.70	429.12	304.82	296.26

We first set $Y0 = \emptyset$, because no branching to 0 occurs in this branching scheme. For each opened bin $i = 1, 2, \ldots, z_1$, if $f_{res}(i) = 0$ we remove from the instance bin i and all the objects it contains. If instead $f_{res}(i) > 0$ we merge all the objects packed in bin i into a unique *composite object*, say k, having weight equal to the sum of the weights of the objects in the bin and fragility equal to that of the witness object in the bin. We then insert object k into set $Y1$. The value of L_2^{01} for the instance just defined is then increased by the number of bins that were removed (because having $f_{res}(i) = 0$). In this way we obtain a valid lower bound that is possibly used to fathom the node.

Since L_2^{01} runs in polynomial time, the resulting B&B algorithm is capable of exploring a large number of nodes in very short computational time, and, as will be described in Section 6, is a good algorithm for small-size BPPFO instances.

5.2. Branch-and-price with branching scheme 3

The third branching scheme we implemented (B&P-3) merges the speed of the B&B (see Section 5.1) with the robustness of column generation. In practice we basically extend the B&B by adding a call to the column generation at any node not fathomed by the lower bound L_2^{01}.

In terms of variables of model (4)–(9), packing object j in a previously opened bin i corresponds to setting $x_{ji} = 1$. Packing j in a new bin of fragility f_j corresponds to setting $y_j = 1$. Hence the results we obtained for B&P-2 are immediately extendible to B&P-3: it is enough to look at rows marked "$y_i = 1$" and "$x_{ji} = 1$" in Table 1.

As happens for the enumeration trees of B&P-1 and B&P-2, also for the tree of B&P-3 any node inherits from the parent node the columns of the restricted master and the additional branching constraints. The negative aspect of B&P-3 is that branching must follow the initial object ordering, hence does not exploit the structure of the fractional solution found at a node. The positive aspect is that we can solve the slave problem with any algorithm of Section 3, in any node of the enumeration tree.

6. Computational results

We coded the algorithms in C++ and tested them on the benchmark set proposed by Clautiaux et al. [2]. This set is publicly available at http://www.or.unimore.it/resources.htm, and consists of 675 instances with 50, 100 and 200 objects, divided into five classes according to the way in which weights and fragilities were created. This benchmark set is particularly challenging, because it derives from a long series of computational tests. A large number of instances were created according to different criteria and computationally tested, and then only the five most difficult classes of instances were kept and made publicly available. For the tests we used an Intel Xeon 2.40 GHz with 6 GB of memory. All LP and ILP models were solved running Cplex 12.2 in sequential mode. In Section 6.1 we present the study we made to find the best B&P configuration. In Section 6.2 the obtained B&P configuration is compared with other exact algorithms for the BPPFO.

6.1. Setting and evaluation of the branch-and-price algorithms

We need to evaluate the impact of the branching scheme, of the algorithm used to solve the slave, and of the use of the additional lower bound L_2^{01}. To this aim we run different configurations of our algorithms with a limited time limit of 900 CPU seconds per instance.

In Table 2 we present the results obtained by running the three branching schemes of Sections 4.1, 4.2 and 5.2, with the three different algorithms to solve the slave. The table presents aggregate information on the complete set of 675 instances: line "gap" gives the average percentage gap between the upper and lower bound provided by the algorithm, line "*opt*" gives the total number of proven optimal solutions, and line "sec" gives the average CPU seconds (obtained by summing the CPU seconds spent for all instances, including those halted at the time limit, and dividing by 675). The best values of *opt*, for each branching scheme, are highlighted in bold.

All algorithms were initialized by computing lower bound L_2, see Algorithm 1, and an upper bound given by a 150 s run of the VNS heuristic of Clautiaux et al. [2], see Section 2. This initialization phase, whose results are given in the second column of the table, solves 312 instances to optimality in an average time of about 82 s, achieving an average gap of 2.27%. The initial lower and upper bounds from [2] are thus quite effective, because they manage to solve almost half of the test instances to proven optimality. (Note that in classical BPP the number of instances that are not solved by upper and lower bounds, for the known benchmarks, is smaller than 10%, thus we can repute our instances a challenging test bed.)

Table 3
Impact of L_2^{01} on B&P-2 and B&P-3.

	B&P-2 (DP)		B&P-3 (DP)	
	Without L_2^{01}	With L_2^{01}	Without L_2^{01}	With L_2^{01}
Gap	0.94%	0.91%	0.80%	0.78%
Opt	*469*	***473***	*492*	***495***
Sec	328.72	313.70	300.43	296.26

The other three groups of columns in the table give the results for the three B&P algorithms, using the three possible configurations for the slave, namely: ILP (see Section 3.1), KP01-based (see Section 3.2) and DP (see Section 3.3). When a slave algorithm is not available for a certain branch decision, we use the previous algorithm. In details:

- B&P-1 was run three times, using, respectively, ILP, KP01-based and DP; in the nodes in which KP01-based or DP were not applicable, the ILP was used instead;
- B&P-2 was run using the three configurations described in the last three columns of Table 1; in the nodes in which DP was not applicable, the KP01-based-$x0$ was used instead;
- B&P-3 was run using the three configurations described in the rows marked "$y_i = 1$" and "$x_{ji} = 1$" in the last three columns of Table 1.

In all cases the restricted masters were initialized with n columns, each containing a single object j, plus the columns belonging to the best heuristic solution found by the VNS.

The results show a similar behavior for the three branching schemes. The KP01-based algorithm improves considerably the results obtained by solving the ILP model with Cplex, leading to larger number of optimal solutions, smaller gap and CPU time. The DP, in turn, improves the results obtained with the KP01-based algorithm. In this case the improvement is smaller, but consistent on all branching schemes. On the basis of these results, in the remaining tests all algorithms were run using the DP.

In Table 2 note also that the average CPU effort is much smaller than the given time limit, and this shows that if a proven optimal solution is found, then it is usually found in quick time. This is due to the good performance of the initial heuristics and of the continuous relaxation of model (10)–(12). Indeed, considering the 495 optimal solutions found by B&P-3 with the DP, the proven optimal value is equal to the initial heuristic upper bound in 428 cases, and is equal to the rounded-up value of the continuous relaxation in 437 cases.

All tests for B&P-2 and B&P-3 summarized in Table 2 were run using L_2^{01} to possibly fathom a node before invoking the more expensive column generation lower bound, see Sections 4.3 and 5.2. In Table 3 we compare these results with those obtained without L_2^{01}. Also in this case the table gives aggregate information over the complete set of 675 instances. It can be noted that for both branching schemes the use of L_2^{01} leads to small but consistent improvements: it decreases the average gap and computational effort, and increases the number of instances solved to proven optimality. In the remaining tests all algorithms were run using L_2^{01}.

Table 4 gives more detailed results that we obtained with the three best performing configurations for the three branching schemes. The first column gives the number of objects ("n") and the index of the class ("cl"). For each value of n and cl, each line gives aggregate information over 45 instances. Lines "avg/*tot*" give average/total values for the 225 instances having the same value of n, and line "overall" gives average/total values for the complete set of 675 instances.

The root node is the same for the three tested algorithms: computation of a lower bound with L_2, computation of an upper bound with the VNS heuristic, improvement of the lower bound by solving the continuous relaxation of model (10)–(12) through the column generation approach described in Section 4 (using the DP to solve the slave problem). The results of the root node are reported in the first group of columns. Apart from "gap", "*opt*" and "sec", that have the same meaning of the previous tables, we also report in "col.s" the average number of columns contained in the set of patterns obtained at the end of the column generation. The root node can solve to proven optimality 371 instances in an average time of about 86 s, leading to an average gap of 1.69%. The average number of columns is usually very small for instances with just 50 objects, but rises consistently for those with 200 objects.

For the three branching schemes we report, respectively, "gap", "*opt*" and "sec", followed by the average number of nodes explored by the enumeration tree ("nodes") and the average number of calls to the DP per node ("calls"). The average values of nodes and calls are evaluated only with respect to the instances that are tackled by the branching schemes, i.e., they do not consider the 371 instances closed to optimality by the root node.

B&P-1 is not effective in solving the small-size instances ($n = 50$) where it is dominated by both B&P-2 and B&P-3. Also for the medium-size instances its behavior is not satisfactory compared to that of B&P-2 and B&P-3. On the other side, for $n = 200$ B&P-1 has the best computational performance on all the five classes, reaching 109 optima against the 83 by B&P-2 and the 77 by B&P-3. B&P-1 usually explores a small number of nodes (225 on average), and performs a small number of calls to the DP (80 on average).

B&P-3 has a behavior that is somehow the opposite to that of B&P-1. It has a very good performance on the small-size instances, all solved to optimality with an additional effort with respect to the root node of just a few seconds. It still has a very good performance on the medium-size instances (193 optima against the 163 of B&P-1 and 167 of B&P-2), but has a

Table 4
Comparison among B&P algorithms (900 s time limit).

n	cl	Root node				B&P-1 (DP)					B&P-2 (DP)					B&P-3 (DP)				
		Gap (%)	*Opt*	Sec	Col.s	Gap (%)	*Opt*	Sec	Nodes	Calls	Gap (%)	*Opt*	Sec	Nodes	Calls	Gap (%)	*Opt*	Sec	Nodes	Calls
50	1	1.25	*38*	58.36	90	0.51	*42*	115.09	268	89	0.00	***45***	59.15	227	99	0.00	***45***	58.62	166	84
	2	0.69	*42*	10.11	27	0.00	***45***	42.80	393	6	0.00	***45***	10.18	67	16	0.00	***45***	10.43	1 057	11
	3	1.82	*35*	47.47	78	1.33	*38*	166.39	225	53	0.17	*44*	68.58	7 516	53	0.00	***45***	47.67	160	51
	4	1.67	*36*	43.40	73	1.17	*39*	162.29	364	49	0.00	***45***	58.72	5 401	53	0.00	***45***	45.25	1 340	52
	5	0.36	*43*	13.41	29	0.36	*43*	46.75	267	76	0.19	*44*	31.18	17 393	73	0.00	***45***	17.18	8 284	78
Avg/*tot*		1.16	*194*	34.55	59	0.67	*207*	106.66	303	55	0.07	*223*	45.56	6 121	59	0.00	***225***	35.83	2 201	55
100	1	1.81	*22*	104.82	336	1.03	***33***	305.17	43	112	1.02	***33***	306.24	11 502	614	1.09	***33***	334.40	28 381	95
	2	1.26	*35*	49.93	204	1.03	*37*	197.18	107	130	0.59	*40*	135.16	6 891	759	0.47	***41***	120.86	18 298	119
	3	1.94	*24*	99.23	295	1.12	*33*	311.88	663	83	0.81	*36*	258.71	12 004	248	0.47	***40***	206.44	15 045	67
	4	1.97	*24*	102.14	307	1.25	*32*	326.69	66	98	1.60	*28*	391.27	23 163	254	0.42	***41***	204.10	17 591	84
	5	2.31	*24*	98.47	357	1.90	*28*	394.94	233	119	1.61	*30*	359.68	14 892	228	0.81	***38***	280.19	20 059	104
Avg/*tot*		1.86	*129*	90.92	300	1.26	*163*	307.17	222	108	1.12	*167*	290.21	13 690	421	0.65	***193***	229.20	19 875	94
200	1	2.25	*7*	139.04	945	1.07	***21***	550.88	59	62	1.24	*19*	572.88	6 652	1981	2.10	*8*	747.20	47 432	23
	2	1.69	*19*	108.21	909	1.37	***24***	451.81	161	144	1.58	*21*	496.73	11 542	878	1.55	*21*	503.59	24 242	88
	3	2.24	*5*	142.67	953	0.98	***25***	484.83	91	46	1.70	*12*	702.84	5 899	524	1.63	*15*	638.55	33 376	7
	4	2.18	*5*	146.30	996	1.38	*17*	636.55	218	59	1.59	*14*	659.81	8 264	824	1.48	***18***	607.62	28 336	35
	5	1.91	*12*	125.46	914	1.26	***22***	526.40	211	72	1.56	*17*	594.34	10 091	1970	1.74	*15*	621.75	30 937	38
Avg/*tot*		2.05	*48*	132.34	943	1.21	***109***	530.09	148	77	1.54	*83*	605.32	8 489	1236	1.70	*77*	623.74	32 865	38
Overall		1.69	*371*	85.93	434	1.05	*479*	314.64	225	80	0.91	*473*	313.70	9 434	572	0.78	***495***	296.26	18 314	62

weak performance for the large-size instances. This behavior may be explained by the fact that using the fractional solution to obtain good branching directions has an important impact only for the large-size instances. B&P-3 explores a very large number of nodes (more than 18 000 on average) and requires a small number of calls to the DP per node (just 62 on average).

B&P-2 is somehow in the middle between B&P-1 and B&P-3: it is better than B&P-1 on the small-size instances and better than B&P-3 on the large-size ones. However it is never the "winning" configuration, and this result may be explained by the very high number of calls to the DP per node that it requires (572 on average). Summarizing, the best B&P configuration that we obtained for the publicly available benchmark set of instances is as follows:

- use DP to solve the slave, and
- if $n \leq 100$ then use B&P-3 with L_2^{01}, otherwise use B&P-1.

This proved to be the best configuration also for computational tests that we performed with larger CPU times.

We conclude this section with a consideration on the effect of the values of the fragilities on the slave solution algorithms. Since the DP algorithm runs in $O(nf_{\max})$, we expect its performance to worsen when the value of $f_{\max}$ gets larger. In the benchmark set of instances, the largest fragility is equal to 450, 750, 750, 450 and 600, for classes 1, 2, 3, 4 and 5, respectively. To estimate how an increase in $f_{\max}$ actually affects the DP, we ran an additional series of tests on the 225 small-size instances having $n = 50$, by running B&P-3 with L_2^{01} and using the three possible configurations for the pricing, and varying weights and fragilities of the items. In particular, in any instance we set $w_j = \mu w_j$ and $f_j = \mu f_j$, for $j = 1, 2, \ldots, n$, and performed three tests by setting μ equal to 1, 10 and 100, respectively. In this way the optimal solutions do not change, but the DP algorithm is forced to perform a larger effort.

The results are depicted in Table 5, where, for each value taken by μ and each B&P-3 configuration, we report the total number of optima, the average gap, the average CPU seconds, and an additional information, sec_{pr}, that gives the average time spent in an instance inside the pricing algorithm. When $\mu = 1$, no change affects the benchmark instances and the DP is the best algorithm. When $\mu = 10$, DP and KP01-based have roughly the same behavior. For $\mu = 100$, the computational effort of the DP increases to almost 4 s on average, and so the KP01-based becomes the most performing algorithm. In any case, both the DP and the KP01-based algorithm outperform the ILP, because the ILP is the only algorithm not capable of solving all instances in the given time limit.

6.2. Comparison among exact algorithms

The best B&P configuration obtained in the previous section is compared in Table 6 with other exact algorithms for the BPPFO, namely: the lower and upper bounding techniques by Clautiaux et al. [2], the mathematical model (4)–(9) of Section 2, and the B&B of Section 5.1. All the algorithms are initialized with L_2 followed by 150 s of the VNS heuristic, as discussed in the previous section. As already noted, this initialization solves to proven optimality the 312 "easier" instances (see the second column of Table 2), and let the exact algorithms work on the remaining difficult instances. Following the

Table 5
Impact of larger fragilities on the slave solution algorithms ($n = 50$, 900 s time limit).

μ	ILP				KP01-				DP			
	Gap (%)	*Opt*	Sec	Sec_{pr}	Gap (%)	*Opt*	Sec	Sec_{pr}	Gap (%)	*Opt*	Sec	Sec_{pr}
1	0.20	220	87.72	49.99	0.00	225	36.63	0.21	0.00	225	36.09	0.05
10	0.20	220	83.82	46.12	0.00	225	37.05	0.24	0.00	225	36.62	0.33
100	0.12	222	84.29	46.65	0.00	225	37.01	0.23	0.00	225	40.25	3.97

Table 6
Comparison among exact algorithms (model (4)–(9), B&B and Best B&P are initialized with L_2 + VNS and run for 1 h time limit).

n	cl	Clautiaux et al. [2]			Model (4)–(9)			B&B			Best B&P		
		Gap (%)	*Opt*	Sec	Gap (%)	*Opt*	Sec	Gap (%)	*Opt*	Sec	Gap (%)	*Opt*	Sec
50	1	1.11	*39*	42.22	0.00	**45**	84.08	0.16	*44*	135.75	0.00	**45**	58.62
	2	0.91	*41*	23.93	0.00	**45**	11.37	0.00	**45**	10.12	0.00	**45**	10.43
	3	1.82	*35*	61.55	0.17	*44*	180.57	0.00	**45**	48.00	0.00	**45**	47.67
	4	1.52	*37*	50.60	0.00	**45**	119.66	0.00	**45**	47.22	0.00	**45**	45.25
	5	0.58	*42*	20.13	0.19	*44*	94.01	0.00	**45**	15.19	0.00	**45**	17.18
Avg/*tot*		1.19	*194*	39.68	0.07	*223*	97.94	0.03	*224*	51.26	0.00	**225**	35.83
100	1	1.81	*23*	236.92	1.20	*31*	1389.37	1.54	*28*	1485.48	0.72	**37**	927.53
	2	1.25	*35*	145.54	1.15	*36*	790.43	0.22	**43**	204.47	0.36	*42*	306.83
	3	1.84	*25*	263.95	1.66	*27*	1733.03	1.18	*32*	1232.08	0.29	**42**	418.63
	4	1.97	*24*	299.91	1.69	*27*	1640.71	1.11	*32*	1233.55	0.22	**43**	385.22
	5	2.50	*22*	244.88	1.70	*29*	1437.53	0.71	*38*	761.37	0.55	**40**	621.10
Avg/*tot*		1.88	*129*	238.24	1.48	*150*	1398.21	0.95	*173*	983.39	0.43	**204**	531.86
200	1	1.92	*9*	770.46	2.07	*6*	3120.01	2.61	*7*	3043.36	1.03	**22**	1922.27
	2	1.68	*18*	728.39	1.82	*17*	2240.20	1.47	*22*	1858.46	1.37	**24**	1712.02
	3	2.16	*6*	940.26	2.60	*6*	3144.99	2.87	*6*	3123.46	0.87	**27**	1587.09
	4	2.06	*4*	932.73	2.28	*4*	3280.12	2.32	*10*	2838.59	1.24	**20**	2134.94
	5	1.92	*11*	685.12	2.19	*11*	2721.63	2.05	*14*	2491.89	1.12	**25**	1806.98
Avg/*tot*		1.95	*48*	811.39	2.19	*44*	2901.39	2.26	*59*	2671.15	1.13	**118**	1832.66
Overall		1.67	*371*	363.11	1.25	*417*	1465.85	1.08	*456*	1235.27	0.52	**547**	800.12

setting used in [2] we run the mathematical model, the B&B and the best B&P with a time limit of 1 CPU hour on each instance. This ensures a fair comparison with the lower and upper bounds in [2], that require at most 3200 s on the same PC that we use in this work.

The algorithms in [2] are fast but obtain just 371 optima and an average gap of 1.67%. Solving model (4)–(9) with Cplex improves the overall number of optima to 417. Note that in this case the use of L_2 and VNS in initialization is very beneficial for the model, because otherwise Cplex would produce just 330 proven optima. Note also that for the large instances ($n = 200$) Cplex is dominated by the lower and upper bounds by [2] for what concerns both gap, time and number of optima.

The B&B we propose produces much better results than the two previous approaches. It solves to optimality all instances but one among the ones with 50 objects, with a computational effort that is less than one minute one average. It also solves 173 instances with 100 objects and 59 with 200 objects, but in these cases it needs quite a large average time. In total it solves 456 instances, but still the resulting average gap that it produces is above 1%.

The B&P is the algorithm that has the best computational performance on average. With the single exception of the group having $n = 100$ and $cl = 2$ (where it is outperformed by the B&B), it always obtains the highest number of proven optima. Notably, it solves to optimality all instance with $n = 50$ in about half a minute and obtains an average percentage gap of just 0.52%. For what concerns the behavior of the algorithms on each class, we notice that class 2 is the easiest one for the first three approaches, whereas class 3 is the easiest one for the B&P. The B&P shows the most stable behavior because it is the only algorithm that consistently solves more than 100 instances for each class.

7. Conclusions

The Bin Packing Problem with Fragile Objects is a difficult combinatorial problem that models, among others, the allocation of cellular calls to frequency channels in telecommunications. In this paper we proposed a branch-and-bound and several branch-and-price algorithms for the exact solution of the problem, and improved their behavior by the use of tailored optimization techniques. During the process we also developed efficient algorithms for the knapsack problem with fragile objects. The computational evaluation that we made on the benchmark set of instances indicates that our algorithms perform very well. Still, several instances with 100 and 200 objects are unsolved to proven optimality, hence future research on this problem is important.

References

[1] N. Bansal, Z. Liu, A. Sankar, Bin-packing with fragile objects, in: R.A. Baeza-Yates, U. Montanari, N. Santoro (Eds.), IFIP TCS, in: IFIP Conference Proceedings, vol. 223, Kluwer Academic Publishers, 2002, pp. 38–46.
[2] F. Clautiaux, M. Dell'Amico, M. Iori, A. Khanafer, Lower and upper bounds for the bin packing problem with fragile objects, Discrete Applied Mathematics (2012) http://dx.doi.org/10.1016/j.dam.2012.04.010. in press.
[3] J.M. Valério de Carvalho, LP models for bin packing and cutting stock problems, European Journal of Operational Research 141 (2002) 253–273.
[4] F. Clautiaux, C. Alves, J.M. Valério de Carvalho, A survey of dual-feasible and superadditive functions, Annals of Operations Research 179 (2010) 317–342.
[5] E.G. Coffman Jr., J. Csirik, G. Galambos, S. Martello, D. Vigo, Bin packing approximation algorithms: survey and classification, in: D.-Z. Du, P.M. Pardalos, R.L. Graham (Eds.), Handbook of Combinatorial Optimization, second ed., Springer, 2013.
[6] C. Alves, J.M. Valério de Carvalho, A stabilized branch-and-price-and-cut algorithm for the multiple length cutting stock problem, Computers & Operations Research 35 (4) (2008) 1315–1328.
[7] F. Vanderbeck, Branching in branch-and-price: a generic scheme, Mathematical Programming, Series A 130 (2011) 249–294.
[8] W.T. Chan, F.Y.-L. Chin, D. Ye, G. Zhang, Y. Zhang, Online bin packing of fragile objects with application in cellular networks, Journal of Combinatorial Optimization 14 (2007) 427–435.
[9] N. Bansal, Z. Liu, A. Sankar, Bin-packing with fragile objects and frequency allocation in cellular networks, Wireless Networks 15 (2009) 821–830.
[10] J.M. Valério de Carvalho, Using extra dual cuts to accelerate column generation, INFORMS Journal on Computing 17 (2) (2005) 175–182.
[11] F. Clautiaux, C. Alves, J.M. Valério de Carvalho, J. Rietz, New stabilization procedures for the cutting stock problem, INFORMS Journal on Computing 23 (2011) 530–545.
[12] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons, Chichester, 1990.
[13] S. Martello, D. Pisinger, P. Toth, Dynamic programming and strong bounds for the 0-1 knapsack problem, Management Science 45 (1999) 414–424.
[14] P.C. Gilmore, R.E. Gomory, A linear programming approach to the cutting stock problem, Operations Research 9 (1961) 849–859.
[15] P.C. Gilmore, R.E. Gomory, A linear programming approach to the cutting stock problem—part II, Operations Research 11 (1963) 863–888.
[16] A.E. Fernandes Muritiba, M. Iori, E. Malaguti, P. Toth, Algorithms for the bin packing problem with conflicts, INFORMS Journal on Computing 22 (2010) 401–415.
[17] U. Pferschy, J. Schauer, The knapsack problem with conflict graphs, Journal of Graph Algorithms and Applications 13 (2009) 233–249.