An optimization algorithm for the ordered open-end bin packing problem

Alberto Ceselli, Giovanni Righini
Dipartimento di Tecnologie dell'Informazione,
Università degli Studi di Milano
Via Bramante 65, 26013 Crema, Italy *

December 14, 2005

Abstract

The ordered open-end bin packing problem is a variant of the bin packing problem in which the items to be packed are sorted in a given order and the capacity of each bin can be exceeded by the last item packed into the bin. We present a branch-and-price algorithm for its exact optimization. The pricing subproblem is a special variant of the binary knapsack problem, in which the items are ordered and the last one does not consume capacity. We present a specialized optimization algorithm for this subproblem. The speed of the column generation algorithm is improved by subgradient optimization steps, allowing for multiple pricing and variable fixing. Computational results are presented on instances of different size and items with different correlations between their size and their position in the given order.

1 Introduction

The open-end bin packing problem is a variant of the bin packing problem [4] in which the capacity of each bin can be exceeded by one of the items packed into the bin. This problem was introduced in a paper by Leung, Dror and Young [7], where the authors proved that the problem is $\mathcal{NP}$-hard. We study a variant of this problem introduced by Yang and Leung [11], called ordered open-end bin packing problem (OOEBPP), in which the items are sorted and the last item in each bin is allowed to exceed the bin capacity. The motivation given by the authors for studying this problem is related to the fare payment in subway stations in Hong Kong. Yang and Leung [11] examined several algorithms for on-line and off-line approximation and studied their worst-case and average-case performance.

* *Correspondence to*: {ceselli,righini}@dti.unimi.it

In this work we present a branch-and-price algorithm for the exact optimization of the OOEBPP. In Section 2 we introduce the notation used in the paper and we give a 0-1 linear programming formulation of the problem; then we present a set covering formulation of the OOEBPP, we introduce a combinatorial lower bound and we show how to derive a set of valid inequalities that may strengthen the set covering formulation. Since the set covering formulation involves an exponential number of variables, we solve it with column generation; in Section 3 we present an ILP formulation of the pricing subproblem, which is a variation of the binary knapsack problem, in which the items are ordered and the last item does not consume capacity. In Section 4 we present a specialized algorithm that allows to effectively solve the pricing subproblem to optimality, exploiting suitable bounds and domination criteria. In Section 5 we describe a branching strategy, three heuristic algorithms and the columns management techniques adopted in our branch-and-price algorithm. In Section 6 we illustrate how we embedded subgradient optimization into column generation, exploiting dual solutions to obtain faster convergence, multiple pricing and variable fixing. In Section 7 we present our computational results with instances of different size and items with different correlation between their size and their position in the given order.

2 Problem formulation

The OOEBPP is defined as follows: an ordered sequence $\mathcal{N}$ of N items is given; each item $j \in \mathcal{N}$ has a given positive integer weight a_j. The items must be packed into identical bins with a given positive integer capacity b. The objective is to minimize the number of bins, subject to the constraint that the capacity of each bin can be exceeded only by the last item packed into it, where the term "last" is referred to the order of the items in $\mathcal{N}$. Hence the last item in each bin requires only one unit of capacity. In the remainder we call such item the *overflow item* of its bin, and we say that it *initializes* its bin. Throughout the paper, we assume that each item in $\mathcal{N}$ is identified by a positive integer in $[1, \ldots, N]$ corresponding to its position in the sequence. An integer linear programming formulation of the problem is the following.

$$\text{minimize} \quad \sum_{i \in \mathcal{N}} y_i \tag{1}$$

$$\text{s.t.} \quad y_i + \sum_{j>i} x_{ij} = 1 \qquad \forall i \in \mathcal{N} \tag{2}$$

$$\sum_{i<j} a_i x_{ij} \leq (b-1) y_j \qquad \forall j \in \mathcal{N} \tag{3}$$

$$x_{ij} \in \{0,1\} \qquad \forall i < j \in \mathcal{N} \tag{4}$$

$$y_i \in \{0,1\} \qquad \forall i \in \mathcal{N} \tag{5}$$

Each binary variable y_i indicates whether item i is the overflow item in its

bin. Hence the number of bins used is indicated in the objective function (1) by the number of binary variables y_i set to 1. Each binary variable x_{ij} indicates whether item i is assigned to the bin in which the overflow item is item j. Because of the constraint on the order of the items, we only have x_{ij} variables with $i < j$. Constraints (2) impose that each item is assigned to a bin, while capacity constraints (3) impose that the overall weight of the items assigned to a bin, excluding the overflow item, must fit into the bin and must leave at least one capacity unit available for accommodating the overflow item.

A lower bound for the OOEBPP can be obtained from the linear relaxation of (1)–(5), when integrality restrictions (4) and (5) are relaxed. In the next subsections we present two other lower bounds: the first one is obtained from the linear relaxation of a set covering formulation of the OOEBPP; the second one is obtained in a combinatorial way, by relaxing integrality constraints on the assignment variables (4). Both lower bounds dominate the lower bound given by the linear relaxation of (1)–(5) and they do not dominate each other. After presenting the two lower bounds, we show how they can be combined together, by adding valid inequalities obtained from the combinatorial lower bounding algorithm to the set covering formulation.

2.1 A set covering formulation

The branch-and-price algorithm we present in this paper relies on the linear relaxation of a set covering formulation of the OOEBPP.

Consider the set Ω_j defined as follows for each $j \in \mathcal{N}$:

$$\Omega_j = \{(x_{ij}, y_j) | \sum_{i<j} a_i x_{ij} \leq (b-1) y_j, 0 \leq x_{ij} \leq 1, 0 \leq y_j \leq 1\}. \tag{6}$$

Let K_j be the set of the integer points in Ω_j and let $(x_{ij}, y_j)^k$ be the generic integer point of Ω_j. Then each point (x_{ij}, y_j) in the convex hull of Ω_j can be expressed as a convex combination of the integer points in K_j:

$$conv(\Omega_j) = \{(x_{ij}, y_j) | (x_{ij}, y_j) = \sum_{k \in K_j} (x_{ij}, y_j)^k z_k, \sum_{k \in K_j} z_k = 1, \ 0 \leq z_k \leq 1\}. \tag{7}$$

Exploiting equation (7) we obtain by substitution the following formulation of the linear relaxation of the OOEBPP, where all polyhedra Ω_j have been replaced

by their convex hulls:

$$\begin{aligned}
\text{minimize} \quad & \sum_{j\in\mathcal{N}} \sum_{k\in K_j} y_j^k z_k & & \\
\text{s.t.} \quad & \sum_{k\in K_i} y_i^k z_k + \sum_{j>i} \sum_{k\in K_j} x_{ij}^k z_k = 1 & \forall i \in \mathcal{N} & \\
& \sum_{k\in K_j} z_k = 1 & \forall j \in \mathcal{N} & (8)\\
& 0 \le z_k \le 1 & \forall j \in \mathcal{N},\ \forall k \in K_j. &
\end{aligned}$$

This model can be simplified as follows. First of all, we exclude from this linear program all columns corresponding to empty bins, that is columns in which $y_j^k = 0$ and $x_{ij}^k = 0$ for each $i < j$. This is correct because empty bins do not give any contribution to defining an OOEBPP solution. The effect of this simplification is that y variables disappear (they are all equal to 1) and constraints (8) are now rewritten as inequalities:

$$\begin{aligned}
\text{minimize} \quad & \sum_{j\in\mathcal{N}} \sum_{k\in K_j} z_k & & \\
\text{s.t.} \quad & \sum_{k\in K_i} z_k + \sum_{j>i} \sum_{k\in K_j} x_{ij}^k z_k = 1 & \forall i \in \mathcal{N} & (9)\\
& \sum_{k\in K_j} z_k \le 1 & \forall j \in \mathcal{N} & (10)\\
& 0 \le z_k \le 1 & \forall j \in \mathcal{N},\ \forall k \in K_j. &
\end{aligned}$$

For each item j which is not chosen as an overflow item, no column in K_j is basic. The second simplification stems from the observation that an optimal solution must exist, in which no item is chosen more than once as the overflow item of a bin. Therefore constraints (10) are redundant and can be dropped. Hence the remaining model only contains set partitioning constraints (9): in turn these can be replaced by set covering constraints, because it is never convenient to pack an item more than once. After these simplifications we obtain the following set covering model:

$$\begin{aligned}
\text{minimize} \quad & \sum_{j\in\mathcal{N}} \sum_{k\in K_j} z_k & & (11)\\
\text{s.t.} \quad & \sum_{k\in K_i} z_k + \sum_{j>i} \sum_{k\in K_j} x_{ij}^k z_k \ge 1 & \forall i \in \mathcal{N} & (12)\\
& 0 \le z_k \le 1 & \forall j \in \mathcal{N},\ \forall k \in K_j. & (13)
\end{aligned}$$

In this model each variable $z_k,\ k \in K_j$ corresponds to a feasible column, that is a feasible set of items packed into a same bin j. This formulation is at least as tight as the linear relaxation of model (1)–(5), owing to the convexification of constraints (3).

Combinatorial Bounding Algorithm

Input: An ordered set $\mathcal{N}$, a weight a_i $\forall i \in \mathcal{N}$, a capacity b.
Output: A set of overflow items $\mathcal{O}$, a set of critical items $\mathcal{C}$, a lower bound on the number of bins $|\mathcal{O}|$.

begin
  /* *Initialization* */
  $\mathcal{C}$:= $\emptyset$; /* $\mathcal{C}$ *is the ordered set of critical items* */
  $\mathcal{O}$:= $\emptyset$; /* $\mathcal{O}$ *is the set of overflow items* */
  R:= 0; /* R *is the overall residual capacity* */
  $\mathcal{S}$:= $\emptyset$ /* $\mathcal{S}$ *is the set of candidate overflow items* */

  for i:= N **down to** 1 **do**
    $\mathcal{S}$:= $\mathcal{S} \cup \{i\}$
    if $a_i > R$ **then**
      /* *Choose the candidate of maximum weight* */
      Select $j^* \in \text{argmax}_{j \in \mathcal{S}}\{a_j\}$
      $\mathcal{C}$:= $\mathcal{C} \cup \{i\}$; $\mathcal{O}$:= $\mathcal{O} \cup \{j^*\}$; $\mathcal{S}$:= $\mathcal{S} \setminus \{j^*\}$
      R:= $R + (b-1) + a_{j^*}$
    R:= $R - a_i$
  /* *Output* */
  return $\mathcal{C}$, $\mathcal{O}$ and $|\mathcal{O}|$
end

Figure 1: Computation of the combinatorial lower bound.

2.2 A combinatorial lower bound

To achieve a lower bound, we consider here the relaxation of the OOEBPP in which the items can be split into different bins and we solve this relaxation to optimality.

Our algorithm (called CBA for Combinatorial Bounding Algorithm in the remainder) computes a set of overflow items in an optimal fractional packing in the following way: items are considered from N down to 1, in order to comply with the constraint on the ordering; at each iteration we define R to be the overall residual capacity of all the bins already initialized. Whenever an item j is found, whose size is greater than R, a new bin is initialized and item j is inserted into an ordered set $\mathcal{C}$ of *critical* items. The overflow item of the new bin is selected as the largest item among those already packed but not yet used as overflow items, in order to yield the maximum residual capacity for the next iterations.

The pseudo-code of CBA is reported in Figure 1. This algorithm returns a lower bound to the number of necessary bins, together with the additional information on the set $\mathcal{C}$ of critical items. We remark that the elements of set $\mathcal{C}$ are sorted according to their insertion order.

If the set $\mathcal{S}$ of the candidate overflow items is implemented with a heap data structure, the complexity of the algorithm is $O(NlogN)$.

The bound provided by CBA dominates the bound provided by the linear programming relaxation of (1)–(5), because the solution given by CBA is feasible for this linear program, but not necessarily optimal.

An example. We illustrate our algorithm CBA with a small example. Consider an OOEBPP instance with 5 items of size 16, 40, 40, 45 and 50 and a bin capacity equal to 50. For each iteration of CBA, starting from item 5 down to item 1, we report in Figure 2 the considered item, its size, the number of bins currently used, the overall residual capacity R, the set of overflow items $\mathcal{O}$ the set of critical items $\mathcal{C}$ and the set of candidate overflow items $\mathcal{S}$.

Note that, when item 3 is considered, the residual capacity is equal to 4 and it is not enough to accommodate the item. Hence a second bin is initialized: its overflow item is item 4, while the current item, that is item 3, is labeled as critical and it is inserted into the set $\mathcal{C}$. The final solution uses only two bins and it is not feasible: the optimal value for this instance (with indivisible items) is 3.

2.3 Combining the two lower bounds

The information given by the critical items computed in algorithm CBA illustrated in subsection 2.2 allows to strengthen the set covering formulation presented in subsection 2.1 by the following valid inequalities:

$$\sum_{k \in \cup_{i \geq \mathcal{C}(t)} K_i} z_k \geq t \quad \forall t = 1, \ldots, |\mathcal{C}|. \tag{14}$$

where $\mathcal{C}(t)$ indicates the t^{th} item that has been inserted in $\mathcal{C}$ according to the insertion order. These inequalities state that at least t overflow items must be selected in the range $[\mathcal{C}(t), \ldots, N]$.

In the example above we have $\mathcal{C}(1) = 5$ and $\mathcal{C}(2) = 3$ and we can add to the master problem the two inequalities:

$$\sum_{k \in K_5} z_k \geq 1 \quad \text{and} \quad \sum_{k \in K_3 \cup K_4 \cup K_5} z_k \geq 2.$$

The first inequality is trivial, since the last item is always an overflow item. The second inequality states that at least two items in the range $[3, \ldots, 5]$ must be overflow items and it may cut off some fractional solutions of the linear relaxation of (11) - (13).

Item	Size	Bins	R	$\mathcal{O}$	$\mathcal{C}$	$\mathcal{S}$
5	50	1	49	$\{5\}$	$\{5\}$	
4	45	1	4	$\{5\}$	$\{5\}$	4
3	40	2	58	$\{4, 5\}$	$\{5, 3\}$	3
2	40	2	18	$\{4, 5\}$	$\{5, 3\}$	2, 3
1	16	2	2	$\{4, 5\}$	$\{5, 3\}$	1, 2, 3

Figure 2: An example of lower bound computation with algorithm CBA.

3 Column generation and the pricing subproblem

The sets K_j of feasible columns have exponentially many elements; therefore a restricted set covering problem (RSCP) is considered and additional columns with negative reduced cost are iteratively generated. For the definition (6) of Ω_j for each given $j \in \mathcal{N}$ the pricing problem we need to solve to generate a new column is a binary knapsack problem. Thus a negative reduced cost column can be generated by solving at most N binary knapsack problems. However solving a large number of knapsack problems to optimality can be unnecessary, since we just need one negative reduced cost column, provided it exists. Therefore we solve a pricing problem in which the overflow item is not fixed, but rather it must be chosen in an optimal way; in other words we search for the column of minimum reduced cost for all possible choices of the overflow item. To formulate the pricing problem we introduce the non-negative dual variables λ associated to covering constraints (12) and the non-negative dual variables μ associated to valid inequalities (14). In the following model each binary variable y_i is equal to 1 if and only if item i is assigned to the bin and it is the overflow item, while each binary variable x_{ij} is equal to 1 if and only if item i is assigned to the bin and item j is the overflow item.

$$
\begin{aligned}
\text{minimize} \quad & \pi(\lambda,\mu) = 1 - \sum_{i\in\mathcal{N}} \lambda_i (y_i + \sum_{j>i} x_{ij}) - \sum_{i\in\mathcal{N}} y_i \sum_{t|\mathcal{C}(t)\le i} \mu_t \\
\text{s.t.} \quad & \sum_{i<j} a_i x_{ij} \le (b-1) y_j && \forall j \in \mathcal{N} \\
& \sum_{i\in\mathcal{N}} y_i = 1 \\
& y_i \in \{0,1\} && \forall i \in \mathcal{N}, \\
& x_{ij} \in \{0,1\} && \forall j \in \mathcal{N},\ \forall i < j.
\end{aligned}
$$

After defining $\rho_i = \lambda_i + \sum_{t|\mathcal{C}(t)\le i} \mu_t$, the pricing problem can be rewritten in an equivalent way, where each binary variable x_i is equal to 1 if and only if item i is assigned to the bin and it is not the overflow item.

$$
\begin{aligned}
\text{minimize} \quad & \pi(\lambda,\mu) = 1 - \sum_{i\in\mathcal{N}} (\lambda_i x_i + \rho_i y_i) && && (15) \\
\text{s.t.} \quad & \sum_{i\in\mathcal{N}} a_i x_i \le b - 1 && && (16) \\
& \sum_{i\in\mathcal{N}} y_i = 1 && && (17) \\
& x_i + \sum_{j\le i} y_j \le 1 && \forall i \in \mathcal{N} && (18) \\
& x_i, y_i \in \{0,1\} && \forall i \in \mathcal{N}.
\end{aligned}
$$

Constraint (16), now involving only the x variables, is the capacity constraint. Constraint (17) states that there must be exactly one overflow item in the bin. Constraints (18) impose the given order to the items: if item i is assigned to the bin and is not the overflow item, then no item j with $j \leq i$ can be the overflow item. Since $\rho_i \geq \lambda_i \ \forall i \in \mathcal{N}$, constraint (17) is implied by constraints (18) and can be dropped.

We call this subproblem the ordered open-end knapsack problem (OOEKP). For analogy with the binary knapsack problem, we state the objective function (15) in maximization form as follows:

$$\text{maximize} \quad \pi'(\lambda, \rho) = \sum_{i \in \mathcal{N}} (\lambda_i x_i + \rho_i y_i).$$

In the next section we present an exact optimization algorithm for the OOEKP.

4 A pricing algorithm

The OOEKP can be solved in $O(Nb)$ computing time via dynamic programming. Let us indicate by $f_{i,w}$ the maximum value of a solution made of items in the range $[1, \ldots, i]$ and consuming w units of capacity. For $i = 1, \ldots, N$ and $w = 0, \ldots, b-1$ the following recursion holds:

$$f_{i,w} = \begin{cases} f_{i-1,w} & \text{if } w < a_i \\ \max\{f_{i-1,w}, f_{i-1,w-a_i} + \lambda_i\} & \text{otherwise} \end{cases}$$

where $f_{0,w} = 0, \forall w = 0, \ldots, (b-1)$. Hence the optimal value of the OOEKP can be found as

$$\pi'(\lambda, \rho) = \max_{i \in \mathcal{N}} \{\rho_i + \max_{w=0,\ldots,b-1} \{f_{i-1,w}\}\}.$$

However this approach is impractical, especially for large size instances with a large value of b.

The algorithm we present here performs an implicit enumeration to identify the optimal overflow item, that is the overflow item of an optimal solution. We present fast bounding techniques and problem reduction tests, coupled with a known effective algorithms for the binary knapsack problem (KP). The worst-case time complexity of our algorithm is worse than that of the dynamic programming approach, since it requires the optimization of a number of KP instances that grows as $O(N)$ in the worst-case. However we experimentally observed that the number of KP instances to be optimized if often very small, and the computing time of our approach is in practice one order of magnitude smaller with respect to the dynamic programming algorithm shown above.

General description. The algorithm initializes a best incumbent lower bound and a set of candidate overflow items $\mathcal{S}$. Then the algorithm computes upper bounds to the value of the OOEKP for each possible choice of the overflow item.

These upper bounds are used both to guide the search in a best-first-search fashion and to terminate the algorithm. The algorithm iteratively selects a most promising overflow item according to its associated upper bound, it solves the corresponding KP instance and this yields a feasible OOEKP solution. The information provided by the optimal solution of the KP instance is also exploited by additional fathoming rules to reduce the number of possible candidate overflow items to be considered.

Preprocessing and initialization. Consider the range $\{1, \ldots, l\}$ such that $\sum_{j=1}^{l-1} a_j \leq b$ and $\sum_{j=1}^{l} a_j > b$. The optimal solution of the OOEKP involving only items in $\{1, \ldots, l\}$ can be computed in linear time because the capacity constraint is not binding when the overflow item is in $[1, \ldots, l]$. This optimal value is kept as an initial lower bound and all items in the range $[1, \ldots, l]$ are no longer considered as candidate overflow items.

Reduction. Some more items that cannot be optimal overflow items are identified as follows. For each pair of items i and j with $i < j$ such that $\rho_i \leq \rho_j$, item i can be discarded from the set $\mathcal{S}$ of candidate overflow items: given a feasible OOEKP solution with i as the overflow item, a non-worse feasible OOEKP solution can be obtained by replacing item i with item j, since feasibility is not affected by the size of the overflow item and the objective function value does not decrease. All items for which this reduction test succeeds are deleted from the candidate set $\mathcal{S}$.

Notation. In the remainder we use the following notation. We indicate with KP_j the optimal value of the binary knapsack problem instance in which the only items available are those in the range $[1, \ldots, j-1]$, and the capacity of the knapsack is equal to $b-1$.

$$KP_j = \max\{\sum_{i=1}^{j-1} \lambda_i x_i : \sum_{i=1}^{j-1} a_i x_i \leq b-1, x_i \in \{0,1\}\ \forall i = 1, \ldots, j-1\}.$$

We also indicate with LKP_j the optimal solution of the linear relaxation of KP_j:

$$LKP_j = \max\{\sum_{i=1}^{j-1} \lambda_i x_i : \sum_{i=1}^{j-1} a_i x_i \leq b-1, 0 \leq x_i \leq 1\ \forall i = 1, \ldots, j-1\}.$$

Finally we indicate with $OOEKP_j$ the optimal value of the ordered open-end knapsack problem in which item j is selected as the overflow item:

$$OOEKP_j = KP_j + \rho_j.$$

Step 1: computation of the upper bounds. The first step of our algorithm consists in computing an upper bound u_j for each possible choice of the overflow item $j \in \mathcal{S}$. For the definitions above, the value

$$u_j = LKP_j + \rho_j$$

is an upper bound to the optimal value of the OOEKP in which j is the overflow item, that is:

$$u_j \geq OOEKP_j.$$

The computation of each upper bound u_j requires the optimization of a continuous KP instance, which can be carried out in $O(N)$ time [1]. However, instead of solving N continuous KP instances, the optimal solution of each of them can be obtained by suitably exploiting the structure of the optimal solution of the previous one and this yields a better worst-case computational complexity and a significant reduction in computing time. Consider the *efficiency* of each item i, that is the ratio $e_i = \lambda_i/a_i$, and consider a list $\mathcal{L}$ of the items sorted by non-increasing value of efficiency. This is computed in $O(NlogN)$ time. The optimal solution of a continuous KP instance can be found by selecting items according to the efficiency order, until an item is found whose weight exceeds the residual capacity. In order to fill the knapsack, such item (called *break item*) is taken with a fractional value. In our algorithm we scan the set of candidate overflow items $\mathcal{S}$, starting from item N down to item l and we scan the ordered list $\mathcal{L}$ from the most to the least efficient item; assume LKP_j has been computed and let $i \in \mathcal{S}$ be the next candidate overflow item to be considered; assume $\bar{k}$ is the current break item in the optimal solution of value LKP_j. In the next iteration all items from $j-1$ down to i become unavailable and the corresponding variables are fixed to 0. If some of these variables are basic in the optimal solution of the previous continuous knapsack, this yields some slack capacity available in the knapsack, which can be filled by other items, which are chosen scanning $\mathcal{L}$ from $\bar{k}$ onward. Once $\mathcal{L}$ has been sorted in $O(NlogN)$ time, the worst-case computational complexity of the remaining procedure is $O(N)$, because each element of $\mathcal{S}$ and $\mathcal{L}$ is considered only once.

Step 2: search. In the second step at each iteration the most promising overflow item k is chosen: $k \in \text{argmax}_{j\in\mathcal{S}}\{u_j\}$ where $\mathcal{S}$ is the set of candidate overflow items not yet considered or fathomed. As soon as u_k is found to be not greater than the best incumbent lower bound, the algorithm terminates. Once the most promising item k has been selected, a binary knapsack problem is solved, where the only available items are those with index less than k.

To solve KP instances we used Pisinger's MINKNAP algorithm [10], which is very fast and exploits the optimal solution of the continuous relaxation both as a dual bound and to identify a good starting primal solution. Every time we optimize a KP instance, corresponding to overflow item k, we get an optimal value KP_k: the corresponding optimal solution is a lower bound to $\pi'(\lambda, \rho)$, since it is a feasible solution of the current pricing subproblem instance; moreover it can be exploited to skip the computation of further KP instances. Let this solution be defined as

$$\overline{x} \in \text{argmax}\{\sum_{i=1}^{k-1} \lambda_i x_i | \sum_{i=1}^{k-1} a_i x_i \leq b-1, x_i \in \{0,1\}\ \forall i = 1, \ldots, k-1\}$$

Optimization algorithm for the OOEKP

Input: An ordered set $\mathcal{N}$; for each $j \in \mathcal{N}$, a weight a_j, a prize λ_j for being inserted into the knapsack and a prize ρ_j for being the overflow item; a capacity b.
Output: An optimal OOEKP solution (x^*, y^*) and its value $\pi'(\lambda, \rho)$

begin
  /* *Initialization* */
  $\pi'(\lambda, \rho) := -\infty$; $l := 1$
  while $(\sum_{j=1}^{l-1} a_j \leq b - 1)$ **do**
    if $(\sum_{j=1}^{l-1} \lambda_j + \rho_l > \pi'(\lambda, \rho))$ **then**
      $\pi'(\lambda, \rho) := \sum_{j=1}^{l-1} \lambda_j + \rho_l$
      $x_j^* := 1 \ \forall j < l$; $x_j^* := 0 \ \forall j \geq l$
      $y^* := 0$; $y_l^* := 1$
    $l := l + 1$
  $\mathcal{S} := \{l + 1, \ldots, N\}$

  /* *Reduction* */
  for each $j \in \mathcal{S}$ **do**
    for each $i \in \mathcal{S},\ i < j$ **do**
      if $(\rho_i \leq \rho_j)$ **then** $\mathcal{S} := \mathcal{S} \setminus \{i\}$

  /* *Compute upper bounds from linear relaxations* */
  for each $j \in \mathcal{S}$ **do** $u_j := \rho_j + LKP_j$

  /* *Examine all candidate overflow items* */
  repeat
    /* *Select the most promising candidate* */
    Select $k \in \text{argmax}_{j \in \mathcal{S}}\{u_j\}$
    /* *Termination test* */
    if $(u_k \leq \pi'(\lambda, \rho))$ **then goto end**
    /* *Solve a KP, store the optimal solution and its value* */
    Select $\overline{x} \in \text{argmax}\{\sum_{i=1}^{k-1} \lambda_i x_i : \sum_{i=1}^{k-1} a_i x_i \leq b - 1, x_i \in \{0, 1\}\ \forall i = 1, \ldots, k - 1\}$
    $KP_k := \sum_{i=1}^{k-1} \lambda_i \overline{x}_i$
    for each $j = k, \ldots, N$ **do** $\overline{x}_j := 0$

    /* *Identify the best overflow item* */
    $h := k$
    while $(\overline{x}_h = 0)$ **do**
      if $(KP_k + \rho_h > \pi'(\lambda, \rho))$ **then**
      /* *Update the best incumbent primal solution* */
        $\pi'(\lambda, \rho) := KP_k + \rho_h$;
        $x^* := \overline{x}$
        $y^* := 0$; $y_h^* := 1$
      $\mathcal{S} := \mathcal{S} \setminus \{h\}$; $h := h - 1$
  until $(\mathcal{S} = \emptyset)$
end

Figure 3: Pseudo-code of the OOEKP optimization algorithm

and let $h = \max\{i|x_i = 1\}$ be the first non-zero component in $\overline{x}$. Then for each $h < j \leq k$ the optimal value $OOEKP_j$ is given by $\rho_j + KP_k$ and the items in the range $[h+1, \ldots, k]$ can be discarded from $\mathcal{S}$; in fact, fixing any $y_j = 1$ with $h < j \leq k$ would not change the optimal solution of the binary KP instance.

The pseudo-code of the pricing algorithm is reported in Figure 3.

5 Branch-and-price

In this section we describe our branch-and-price algorithm: we illustrate our branching strategy, we describe how primal bounds are obtained at the root node and possibly improved at each node of the search tree by fast heuristics and we describe the techniques we use to manage the pool of columns.

5.1 Branching strategy

Our branching rule is based on the variables of the original formulation (1) - (5). Once an optimal solution z^* of the master problem is obtained, a corresponding (fractional) solution (x^*, y^*) in terms of the original variables can be reconstructed through the relations $x^*_{ij} = \sum_{j>i} \sum_{k \in K_j} x^k_i z^*_k$ and $y^*_j = \sum_{k \in K_j} z^*_k$ for each $i, j \in \mathcal{N}$.

We have adopted a two levels branching strategy: in the first level search tree the branching decisions are taken on the y_j variables, that is the overflow items of the bins are chosen; the variable y_j whose current optimal value y^*_j is closest to 0.5 is selected and two branches are considered: j is discarded from the set of candidate overflow items in the first branch and it is selected as an overflow item in the second branch. In the second level search tree, where the number of bins has been defined and the overflow item of each bin has been chosen, we are left with a feasibility problem. In this second level search tree we use a binary branching rule similar to the previous one, selecting the x_{ij} variable whose current optimal value x^*_{ij} is closest to 0.5.

Fixing an x_{ij} variable is easily taken into account at pricing level, because it just reduces the dimension of the pricing subproblem. Fixing a y_j variable to 0 is also easy to manage: the item j is simply dropped from the set $\mathcal{S}$ of candidate overflow items. When a y_j variable is fixed to 1, two cases must be taken into account at pricing level: each time we solve the pricing subproblem, either j is not included in the optimal OOEKP solution or j must be the overflow item; therefore, we first exclude j and solve the remaining OOEKP; then we fix j as the overflow item and solve the remaining KP; finally we take the best of these two solutions. Owing to the multiple choice constraint (17) this does not provoke a combinatorial explosion in the number of cases to consider: in a node of the search tree where n binary y variables have been fixed to 1, we need to consider $n + 1$ cases: either one of them is the overflow item and the others are disregarded or all of them must be disregarded.

The search tree is explored with a best-bound-first policy.

5.2 Primal bounds

We have used three different heuristic algorithms to compute primal feasible solutions to the OOEBPP quickly.

The first one is an adaptation of the well-known Best-Fit Decreasing-Height (BFDH) approximation algorithm [8], which we indicate as Best-Fit Decreasing-Time (BFDT). The items are iteratively considered from item N down to item 1 and at each iteration the current item is packed into the bin with the minimum residual capacity among those which can accommodate it; if no bin can receive the item, a new bin is initialized. The pseudo-code of this algorithm is reported in Figure 4.

Best-Fit Decreasing-Time heuristic

Input: An ordered set $\mathcal{N}$; a weight a_i for each $i \in \mathcal{N}$; a capacity b.
Output: The set of overflow items $\mathcal{O}$ in a feasible OOEBPP solution, and the corresponding value $|\mathcal{O}|$.

begin
  /* *Initialization* */
  $\mathcal{O}$:= $\emptyset$; $\mathcal{S}$:= $\emptyset$
  for $i \in \mathcal{N}$ **do** $J(i)$:= $\emptyset$ /* $J(i)$ *is the set of items in the bin whose overflow item is* i */

  /* *BFDT computation* */
  for i:= N **down to** 1 **do**
    $\mathcal{S}$:= $\mathcal{S} \cup \{i\}$
    /*Compute the set of bins in which i can be inserted */
    $F(i)$:= $\{j \in \mathcal{O} | \sum_{k \in J(j)} a_k + a_i \leq b - 1\}$
    if $F(i) = \emptyset$ **then**
      /* *Initialize a new bin with the candidate of largest weight* */
      Select $i^* \in \text{argmax}_{i \in \mathcal{S}}\{a_i\}$;
      $\mathcal{O}$:= $\mathcal{O} \cup \{i^*\}$; $\mathcal{S}$:= $\mathcal{S} \setminus \{i^*\}$
      if $i^* \neq i$ **then**
        /* *Remove* i^* *from its bin; since* $a_i \leq a_{i^*}$ *this becomes* */
        /* *the bin with minimum residual capacity that can receive item* i */
        j^*:= $j \in \mathcal{O} | i^* \in J(j)$;
        $J(j^*)$:= $J(j^*) \cup \{i\} \setminus \{i^*\}$;
    else
      Select $j^* \in \text{argmax}_{j \in F(i)}\{\sum_{k \in J(j)} a_k\}$
      $J(j^*)$:= $J(j^*) \cup \{i\}$

  /* *Output* */
  return $\mathcal{O}$ and $|\mathcal{O}|$
end

Figure 4: Pseudo-code of the Best-Fit Decreasing-Time heuristic.

A second way of computing feasible solutions is a simple randomized version of the BFDT algorithm, called RBFDT, in which a set of r overflow items is drawn from a uniform probability distribution and r corresponding bins are initialized (the only exception is the last item, which is always fixed as an overflow item). We considered values of r ranging from 1 to $\lfloor 0.5 \sum_{j \in \mathcal{N}} a_j / b \rfloor$; this threshold was chosen in order to randomly select at most half of the overflow items. We ran RBFDT 10 times for each value of r.

A third heuristic consists in taking the current fractional solution of the linear relaxation of the master problem and to round up some of the basic z

variables, until all rows are covered by columns associated to variables equal to 1. This rounding is carried out in a greedy way: the z_k variables are considered in order of non-increasing fractional value and each variable is set to 1 if and only if it covers at least one row left uncovered by the columns whose variables have been already set to 1.

The BFDT and RBFDT heuristics are executed once at the root node, as an initialization step of our algorithm and the primal solutions obtained in this way are used to populate the initial RSCP. The two heuristics pursue two different objectives: with BFDT we look for "good" initial solutions, while with RBFDT we generate a diversified initial set of columns of the RSCP.

Instead, the rounding heuristic is executed once for each node of the search tree, when the column generation process is over.

5.3 Columns removal and insertion

We found useful to periodically remove unpromising columns from the RSCP: each time a new node of the search tree is considered, the columns in the RSCP whose reduced costs are greater than a threshold are moved into a pool. To this purpose the reduced cost of each column is computed with respect to the optimal dual solution on the ancestor node. In our implementation the threshold is set to $1/(2N)$, in order to keep the RSCP small.

The pool of columns is scanned at each column generation iteration: whenever a column is found whose reduced cost is negative with respect to the current dual solution, it is moved back into the RSCP. A column is erased from the pool if its reduced cost has been found to be non-negative for 6 consecutive times. This parameter is tuned in accordance to our previous experience with pool management techniques (see for instance [3] and [2]).

6 Lagrangean bounds

The bound obtained by optimizing the set covering formulation of the OOEBPP can also be obtained through Lagrangean relaxation of the set of constraints (2). For the theoretical equivalence between column generation and Lagrangean relaxation, we refer the reader to the recent book by Desaulniers et al. [5]. The exploitation of this equivalence can be very influential on the effectiveness of a column generation algorithm, as shown for instance in reference [3]. In this section we describe an effective way of alternating primal simplex iterations of the column generation procedure with subgradient optimization iterations to locally improve the dual solution: this allows to obtain tight dual bounds quickly and to design effective multiple pricing and variable fixing procedures.

The Lagrangean subproblem originating from the relaxation of constraints

(2) is the following:

$$\begin{aligned}
\text{minimize} \quad & \sum_{j\in\mathcal{N}} y_j - \sum_{i\in\mathcal{N}} \lambda_i (y_i + \sum_{j>i} x_{ij} - 1) & \\
\text{s.t.} \quad & \sum_{i<j} a_i x_{ij} \leq (b-1) y_j & \forall j \in \mathcal{N} \\
& x_{ij} \in \{0,1\} & \forall j \in \mathcal{N},\ \forall i < j \\
& y_j \in \{0,1\} & \forall j \in \mathcal{N}.
\end{aligned}$$

For each set of multipliers $\lambda \geq 0$ the problem decomposes into independent subproblems, one for each $j \in \mathcal{N}$:

$$\begin{aligned}
\text{minimize} \quad & (1-\lambda_j) y_j - \sum_{i<j} \lambda_i x_{ij} & \\
\text{s.t.} \quad & \sum_{i<j} a_i x_{ij} \leq (b-1) y_j & \\
& x_{ij} \in \{0,1\} & \forall i < j \\
& y_j \in \{0,1\}. &
\end{aligned}$$

Each subproblem can be optimized considering two cases according to the value of y_j. For $y_j = 1$ a binary knapsack problem is solved. This is equivalent to fix j as the overflow item and optimize problem (15) – (18). In order to highlight this analogy, we indicate the value obtained in this way by $\pi_j(\lambda)$. If $\pi_j(\lambda) > 0$, a better solution is found by fixing y_j to 0, and by consequently setting x_{ij} to 0 for each $i < j$. Hence, for any choice of the Lagrangean multipliers, a valid lower bound $\omega(\lambda)$ for the OOEBPP is given by

$$\omega(\lambda) = \sum_{i\in\mathcal{N}} \lambda_i + \sum_{j\in\mathcal{N}} \min\{\pi_j(\lambda), 0\}.$$

The main advantage of our pricing method is to avoid the optimization of a large number of KP instances, since in the OOEKP algorithm we implicitly consider all π_j's to identify the one with minimum value. However a lower bound $\underline{\omega}(\lambda)$ to $\omega(\lambda)$ can be obtained by replacing each $\pi_j(\lambda)$ value with a corresponding lower bound $\underline{\pi}_j(\lambda)$.

To this purpose, we exploit the solution of the relaxed OOEKP: we initially set the $\underline{\pi}_j(\lambda)$ values to the u_j bounds, which are readily available from the pricing algorithm; then, whenever KP_j subproblem is solved to optimality during the execution of the pricing algorithm, the corresponding $\underline{\pi}_j(\lambda)$ bound is updated. Finally the information from the combinatorial bound presented in subsection 2.2 can be used to strengthen $\underline{\omega}(\lambda)$. A set of valid inequalities for the original formulation, analogous to constraints (14), is

$$\sum_{i\geq\mathcal{C}(t)} y_i \geq t \qquad \forall t = 1, \ldots, |\mathcal{C}|. \tag{19}$$

Select overflow items:

Input: A set of $\underline{\pi}_j(\lambda)$ values
Output: A set $\mathcal{O}$ of selected overflow items

begin
  $\mathcal{O}$:= $\emptyset$
  for each $t \in \mathcal{C}$ **do**
    Select $j^*(t) \in \text{argmin}_{j \in \mathcal{N} \setminus \mathcal{O}, j \geq \mathcal{C}(t)} \{\underline{\pi}_j(\lambda)\}$;
    $\mathcal{O}$:= $\mathcal{O} \cup \{j^*(t)\}$;
  $\mathcal{O}$:= $\mathcal{O} \bigcup \{j \in \mathcal{N} | \underline{\pi}_j(\lambda) < 0\}$;
end

Figure 5: Computation of the Lagrangean lower bound

A dual bound $\underline{\omega}(\lambda)$ is computed as follows. First, the best set of overflow items is identified, that satisfy constraints (19); then further "desirable" overflow items, that is those with $\pi_j(\lambda) < 0$, are selected. This procedure is detailed in Figure 5. Finally, $\underline{\omega}(\lambda)$ is computed as

$$\underline{\omega}(\lambda) = \sum_{i \in \mathcal{N}} \lambda_i + \sum_{j \in \mathcal{O}} \underline{\pi}_j(\lambda).$$

Whenever, during the column generation iterations, the difference between the largest $\underline{\omega}(\lambda)$ value encountered and the RSCP optimal value is less than 10^{-6}, the column generation process is terminated, and the Lagrangean bound is kept as the final lower bound for the current node of the search tree.

6.1 Multiple pricing

The equivalence with Lagrangean relaxation is exploited also to search for different sets of columns at each column generation iteration. In fact, it is a common practice in Lagrangean relaxation-based algorithms to iteratively improve a dual solution with subgradient optimization [6]. In our case the subgradients are not readily available, since we avoid to optimize several KP subproblems in our pricing algorithm. Therefore, as in the computation of the $\underline{\pi}_j(\lambda)$ values, in all cases in which the exact optimization of a KP subproblem has not been carried out during the execution of the pricing algorithm, we use the fractional solution of the corresponding continuous KP instance to approximate the subgradient.

At each column generation step we initialize a set of Lagrangean multiplier values λ^0 with the current values of the dual variables λ; then we perform at most 50 subgradient iterations, starting with a step parameter value of 2.0 and halving it every 10 iterations in which the lower bound has not been improved; in this way we obtain a sequence of Lagrangean multipliers $\lambda^0, \ldots, \lambda^{50}$.

Let $\lambda^* \in \text{argmax}_{s<g}\{\underline{\omega}(\lambda^s)\}$ be the set of multipliers corresponding to the best incumbent lower bound before a generic subgradient iteration g. Whenever λ^g improves the lower bound given by λ^*, the column corresponding to the OOEKP optimal solution at iteration g is inserted into the RSCP.

In this way for each column generation iteration several subgradient optimization iterations are performed and more than one column can be added to

the restricted master problem. Both subgradient optimization iterations and column generation iterations call the same subroutine, that is the optimization algorithm for the OOEKP presented in Section 4.

This hybrid technique, mixing the primal simplex algorithm with subgradient optimization, yields substantial improvements to the convergence rate of the column generation algorithm.

6.2 Variable fixing

We also used the $\underline{\pi}_j(\lambda^s)$ values in a variable fixing procedure, to evaluate the effect of flipping the binary variables representing the selection of each overflow item.

Consider the set of items $j \geq \mathcal{C}(t)$ for some $t = 1, \ldots, |\mathcal{C}|$. When the corresponding constraint (19) is not active, the effect of flipping a y_j with value 1 (resp. 0) is evaluated by adding (resp. subtracting) the $\underline{\pi}_j(\lambda^s)$ value to (from) the lower bound. On the opposite, when this constraint is active, dropping an overflow item requires the selection of another one in the corresponding interval; in a similar way, the selection of an additional overflow item may allow to drop the least profitable selected one.

Hence, considering a generic iteration s of the subgradient optimization algorithm, for each $t = 1, \ldots, |\mathcal{C}|$, let $\underline{\pi}^{BO}(t)$ be the minimum $\underline{\pi}_j(\lambda^s)$ value among the unselected items $j \in \mathcal{N} \setminus \mathcal{O}, j \geq \mathcal{C}(t)$, and $\underline{\pi}^{WI}(t)$ be the maximum $\underline{\pi}_j(\lambda^s)$ value among the selected items $j \in \mathcal{O}, j \geq \mathcal{C}(t)$ (BO stands for 'Best Out' and WI stands for 'Worst In'). If $|\{j \in \mathcal{O} | j \geq \mathcal{C}(t)\}| > t$, set $\underline{\pi}^{BO}(t) = \underline{\pi}^{WI}(t) = 0$; if $\underline{\pi}^{WI}(t) < 0$ set $\underline{\pi}^{WI}(t) = 0$.

Then for each $t = 1, \ldots, |\mathcal{O}|$

- for each $j \geq \mathcal{C}(t)$ such that $j \in \mathcal{N} \setminus \mathcal{O}$, if $\lceil \underline{\omega}(\lambda^s) + \underline{\pi}_j(\lambda^s) - \underline{\pi}^{WI}(t) \rceil \geq UB$, then j can be discarded from the set of candidate overflow items (y_j is set to 0);
- for each $j \geq \mathcal{C}(t)$ such that $j \in \mathcal{O}$, if $\lceil \underline{\omega}(\lambda^s) - \underline{\pi}_j(\lambda^s) + \underline{\pi}^{BO}(t) \rceil \geq UB$, then j can be fixed as an overflow item (y_j is set to 1).

In both cases flipping the value of the y variable would produce a lower bound not smaller than the best incumbent upper bound.

7 Computational results

We tested our branch-and-price algorithm on two data-sets proposed in the literature for bi-dimensional packing problems. The first data-set is described in [8] and consists of 5 classes of instances: BENG (10 instances), CGCUT (3 instances), GCUT (4 instances), HT (9 instances) and NGCUT (12 instances). The second data-set is described in [9] and consists of 500 instances, divided into 10 classes of 50 instances, named MV and BW. In bi-dimensional packing problems each item has both a width and a height and the aforementioned

data-sets contain instances with different types of correlation between these two parameters. In order to obtain OOEBPP instances, we interpreted the height of each item as a "time-stamp": if item i has a smaller height than item j in the bi-dimensional packing instance, then item i precedes item j in the corresponding OOEBPP instance. To obtain a total order, we broke the ties according to the order given in the original data file.

Our branch-and-price algorithm was implemented in C++. CPLEX 8.1 was used to solve the LP relaxations. The code was compiled with the GNU CC version 3.2.2, with full optimization. Our computational results were obtained on a Linux workstation equipped with a Pentium IV 1.6GHz processor and 512MB of RAM. We imposed to every test a time limit of one hour.

Dual bounds. In a first set of tests, we compared the tightness of the dual bounds described in Section 2. In Tables 1(a) and 1(b) we report our results on the first and the second data-set respectively. Each table is made by five blocks: in the first one we include the class of instances, while each of the subsequent four blocks refers to the dual bounding technique indicated in the leading row. We denote the combinatorial bound with CB, the linear relaxation of the original formulation with LP, the relaxation given by the set covering formulation without constraints (14) with CG and the relaxation given by the set covering formulation with constraints (14) with MIX. Each entry of the tables represents the average value of the instances in a class.

For CB we report the average dual gap, computed as the difference between the optimal value and the value of the bound, divided by the optimal value. We also report the gap when the lower bound is rounded up to the nearest integer, which is always possible since the number of bins in a feasible solution must always be an integer number. We do not report the computation time, because the effort for computing CB and LP for these instances is negligible and the computation of the other two bounds never required more than a few seconds.

The CG bound is always tight: on the first data-set no duality gap was observed when rounding the value of the CG bound up to the nearest integer; on the second data-set, a gap was found on three classes, and it was always smaller than 0.2%. The competitor is CB: it is tighter and faster to compute than LP; it gives rather tight bounds (except for class BW06, where the duality gap is more than 11%). In class MV02 it is better than the CG bound too. It is worth noting that combining CG and CB techniques in the MIX relaxation yields sometimes (e.g. on a set of GCUT instances) a bound that is tighter than the best of the two.

Primal bounds. In Tables 2(a) and 2(b) we report the cost of the feasible solutions found by three heuristics at the root node, for the instances in the first and second data-set respectively. These tables consist of four columns: the first one indicates the class of instances, while the subsequent columns refer to the BFDT, RBFDT and rounding heuristics respectively. Each entry indicates the average gap between the value of the heuristic solution and the optimal value, divided by the optimal value. Randomizing the BFDT heuristic

yields better primal bounds and allows to obtain an initial RSCP with a large enough set of well diversified columns, such that its linear relaxation is rather tight. The rounding heuristic yielded essential improvements only for instances in the class GCUT of the first data-set. We do not report computational times of the heuristics since they were negligible compared to the rest of the algorithm.

Optimal solutions. Finally, we performed a set of tests on the effectiveness of branch-and-price for solving the OOEBPP to optimality, comparing it with CPLEX 8.1 used as an ILP solver.

The branch-and-price algorithm using the combined bound was able to reduce the duality gap very quickly on all instances; however, the relaxed solutions were highly fractional, and it was hard for the heuristics to find the optimal solution. In the most successful version of our method the inequalities (14) were dropped and each μ_t coefficient fixed to 0. Instead, both the CG and CB bounds were computed at each node of the branching tree, and the tightest of them was considered. In this way optimal solutions were found earlier, and less nodes of the branching trees were explored to prove optimality. Therefore we report our computational results only for this last implementation.

The results of our comparison on the first and second data-sets are reported in Tables 3(a) and 3(b). In the first column we indicate the instance class name; then, each table has a block for the results of CPLEX and a block for those of the branch-and-price. For the first data-set we report the average gap between the value of best solution found and the optimal value, divided by the optimal value ("gap"), and the time required to obtain a proven optimal solution ("time"). Both methods completed the computation within the resource limits, but branch-and-price was almost always faster than CPLEX; in particular on classes BENG and GCUT it was two orders of magnitude faster. In Table 3(b) related to the second data-set, we indicate also the number of solved instances in each class ("solved instances"). Branch-and-price solved all the instances but 4, while CPLEX failed on 30 instances. CPLEX exceeded the time limit in 16 cases, and had memory overflow problems in the remaining 14; branch-and-price failures were all due to memory overflow. The remaining instances were solved on the average in less than one minute.

Acknowledgments. The authors acknowledge the kind support of ACSU - Associazione Cremasca Studi Universitari to the OptLab of our department, where this work has been carried out. A preliminary version of this paper was presented at ECCO XVIII in Minsk (May 2005). The work was concluded while the first author was at the Institute for Mathematics, Technical University of Berlin, supported by a Marie Curie fellowship.

References

[1] E. Balas and E. Zemel. An algorithm for large zero-one knapsack problems. *Operations Research*, 28:1130–1154, 1980.

[2] A. Ceselli and G. Righini. A branch-and-price algorithm for the multilevel generalized assignment problem. Technical report, DTI – Università di Milano, 2004.

[3] A. Ceselli and G. Righini. A branch-and-price algorithm for the capacitated p-median problem. *Networks*, 45(3):125 – 142, 2005.

[4] E.G. Coffman Jr., M. R. Garey, and D. S. Johnson. *Approximation algorithms for bin packing: a survey, in Approximation algorithms for NP-hard problems*. D. Hochbaum ed., PWS Publishing Company, Boston, U.S.A., 1996.

[5] G. Desaulniers, J. Desrosiers, and M.M. Solomon, editors. *Column Generation*. Springer, 2005.

[6] M. Held, P. Wolfe, and H.P. Crowder. Validation of subgradient optimization. *Mathematical Programming*, 6:62–88, 1974.

[7] J.Y.T. Leung, M. Dror, and G. H. Young. A note on an open-end bin packing problem. *Journal of Scheduling*, 4:201–207, 2001.

[8] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: a survey. *European Journal of Operational Research*, 141:241–252, 2002.

[9] A. Lodi, S. Martello, and D. Vigo. Models and bounds for two dimensional packing problems. *Journal of Combinatorial Optimization*, 8:363 – 379, 2004.

[10] D. Pisinger. A minimal algorithm for the 0–1 knapsack problem. *Operations Research*, 45:758–767, 1997.

[11] J. Yang and J. Y.T. Leung. The ordered open-end bin-packing problem. *Operations Research*, 51:759–770, 2003.

(a)

	CB	LP		CG		MIX	
	dual gap	ceil dual gap	dual gap	ceil dual gap	dual gap	ceil dual gap	dual gap
BENG	0.00%	0.00%	-7.24%	0.00%	-7.23%	0.00%	0.00%
CGCUT	0.00%	0.00%	-6.26%	0.00%	-3.50%	0.00%	0.00%
GCUT	-5.00%	-5.00%	-11.51%	0.00%	-3.33%	0.00%	-2.50%
HT	0.00%	-2.78%	-11.40%	0.00%	-11.05%	0.00%	0.00%
NGCUT	-4.71%	-6.10%	-20.97%	0.00%	-11.30%	0.00%	0.00%

(b)

	CB	LP		CG		MIX	
	dual gap	ceil dual gap	dual gap	ceil dual gap	dual gap	ceil dual gap	dual gap
MV01	-2.95%	-2.95%	-5.69%	0.00%	-2.78%	0.00%	-1.16%
MV02	0.00%	-0.13%	-7.13%	-0.13%	-7.02%	0.00%	0.00%
MV03	-1.24%	-1.40%	-4.98%	-0.11%	-3.05%	-0.11%	-0.79%
MV04	0.00%	0.00%	-6.68%	0.00%	-6.46%	0.00%	0.00%
BW01	-1.39%	-1.56%	-5.56%	-0.17%	-3.00%	-0.17%	-1.00%
BW02	0.00%	0.00%	-7.22%	0.00%	-7.02%	0.00%	0.00%
BW03	-1.72%	-2.42%	-7.41%	0.00%	-4.99%	0.00%	-1.47%
BW04	-8.99%	-9.37%	-11.98%	0.00%	-1.07%	0.00%	-1.04%
BW05	-11.72%	-11.78%	-14.89%	0.00%	-1.72%	0.00%	-1.72%
BW06	-0.18%	-0.18%	-5.32%	0.00%	-4.80%	0.00%	-0.17%

Table 1: Comparison of dual bounds

(a)

	BFDT gap	RBFDT gap	Rounding gap
BENG	0.00%	0.00%	0.00%
CGCUT	5.88%	1.96%	1.96%
GCUT	8.33%	8.33%	1.67%
HT	0.00%	0.00%	0.00%
NGCUT	0.00%	0.00%	0.00%

(b)

	BFDT gap	RBFDT gap	Rounding gap
MV01	2.59%	1.96%	1.68%
MV02	1.07%	0.89%	0.89%
MV03	6.88%	5.90%	5.57%
MV04	4.28%	4.01%	4.01%
BW01	7.93%	6.58%	5.34%
BW02	2.61%	1.61%	1.61%
BW03	5.06%	3.67%	3.67%
BW04	5.84%	1.95%	1.86%
BW05	6.63%	2.04%	1.55%
BW06	8.10%	6.11%	6.11%

Table 2: Comparison of primal bounds

(a)

	CPLEX		Branch-and-price	
	gap	time(s)	gap	time (s)
BENG	0.00%	1.73	0.00%	0.08
CGCUT	0.00%	0.09	0.00%	0.04
GCUT	0.00%	100.85	0.00%	1.65
HT	0.00%	0.04	0.00%	0.02
NGCUT	0.00%	0.02	0.00%	0.03

(b)

	CPLEX			Branch-and-price		
	solved instances	gap	time (s)	solved instances	gap	time (s)
MV01	47	0.00%	16.75	50	0.00%	2.67
MV02	50	0.00%	0.50	50	0.00%	0.02
MV03	45	0.32%	14.73	50	0.00%	20.01
MV04	50	0.00%	11.53	50	0.00%	9.83
BW01	41	0.75%	45.99	50	0.00%	5.49
BW02	47	0.49%	0.97	50	0.00%	14.05
BW03	48	0.19%	38.91	49	0.08%	50.91
BW04	48	0.00%	7.30	50	0.00%	1.37
BW05	48	0.00%	1.98	50	0.00%	0.42
BW06	46	0.42%	46.09	47	0.48%	29.77

Table 3: Solving the OOEBPP to proven optimality