Discrete Applied Mathematics 111 (2001) 231–262

DISCRETE APPLIED MATHEMATICS

Lower bounds and algorithms for the 2-dimensional vector packing problem

Alberto Caprara, Paolo Toth *

University of Bologna, DEIS, Viale Risorgimento 2, 40136 Bologna, Italy

Received 1 December 1998; revised 4 November 1999; accepted 10 April 2000

Abstract

Given n items, each having, say, a weight and a length, and n identical bins with a weight and a length capacity, the 2-Dimensional Vector Packing Problem (2-DVPP) calls for packing all the items into the minimum number of bins. The problem is NP-hard, and has applications in loading, scheduling and layout design. As for the closely related Bin Packing Problem (BPP), there are two main possible approaches for the practical solution of 2-DVPP. The first approach is based on lower bounds and heuristics based on combinatorial considerations, which are fast but in some cases not effective enough to provide optimal solutions when embedded within a branch-and-bound scheme. The second approach is based on an integer programming formulation with a huge number of variables, whose linear programming relaxation can be solved by column generation, typically requiring a considerable time, but obtaining extensive information about the optimal solution of the problem. In this paper we first analyze several lower bounds for 2-DVPP. In particular, we determine an upper bound on the worst-case performance of a class of lower bounding procedures derived from BPP. We also prove that the lower bound associated with the huge linear programming relaxation dominates all the other lower bounds we consider. We then introduce heuristic and exact algorithms, and report extensive computational results on several instance classes, showing that in some cases the combinatorial approach allows for a fast solution of the problem, while in other cases one has to resort to the huge formulation for finding optimal solutions. Our results compare favorably with previous approaches to the problem.
© 2001 Elsevier Science B.V. All rights reserved.

Keywords: Two-dimensional vector packing problem; Lower bounds; Branch-and-bound; Branch-and-price; Heuristics

1. Introduction

Given n items, the jth having a weight $w_j > 0$ ($j = 1,\ldots,n$), and n identical bins of capacity $c > 0$, the Bin Packing Problem (BPP) calls for packing all the items into the minimum number of bins, subject to the capacity constraint. This problem is

* Corresponding author. Tel.: +39-051-2093028; fax: +39-051-2093073.
E-mail address: ptoth@deis.unibo.it (P. Toth).

known to be NP-hard in the strong sense. Many heuristic algorithms for BPP have been proposed, and particular efforts have gone into studying their worst-case performance; for a survey of these results see Coffman et al. [6]. Lower bounding procedures and exact enumerative algorithms seem to have received significant attention only recently. A famous lower bound was introduced in the early sixties by Gilmore and Gomory [15,16]. Eilon and Christofides [9] and Hung and Brown [20] proposed enumerative algorithms using a trivial lower bound. In [21] Martello and Toth describe a new lower bound and some reduction procedures, which in [22] are embedded into a branch-and-bound algorithm for an exact solution of the problem. Chen and Srivastava [5] study a generalization of the Martello–Toth lower bounding procedure, while Chan et al. [3] analyze the worst-case performance of the lower bound proposed in [15]. Finally, Vance et al. [28], Vance [27], Valerio de Carvalho [26], and Vanderbeck [29] describe column-generation based branch-and-bound algorithms.

A generalization of BPP, introduced by Garey et al. [12], is the so-called m-Dimensional Vector Packing Problem (m-DVPP), in which each item j has m weights $w_j^1, \ldots, w_j^m \geqslant 0$ $(j = 1, \ldots, n)$ (with $\sum_{l=1}^m w_j^l > 0$) and bins have m capacities $c^1, \ldots, c^m > 0$: the items have to be packed into the minimum number of bins so as to satisfy the capacity constraint for each dimension. Namely, for each bin i, letting S be the set of items packed into bin i, the capacity constraints require $\sum_{j \in S} w_j^l \leqslant c^l$ for $l = 1, \ldots, m$. Note that in m-DVPP each dimension is independent of the others, consequently this problem is different from the well-known m-Dimensional BPP, which requires packing hyperrectangles into hypercubes. As for BPP, different heuristic algorithms of the greedy type have been proposed for m-DVPP, and the worst-case performance of some of them has been analyzed (see [12,31,11,24]). Unlike BPP, for which even simple heuristic algorithms guarantee a good worst-case performance, no polynomial algorithm for m-DVPP is known that gives a solution value which is less than a constant times the optimum in the worst case. In particular, Fernandez de la Vega and Lueker [11] adapt their approximation scheme for BPP to m-DVPP, leading to an algorithm having a worst-case performance ratio of $m + \varepsilon$, for any fixed $\varepsilon > 0$. The running time is linear in n, but, unfortunately, exponential in $1/\varepsilon^2$. Furthermore, one has to solve a linear program which is huge in size for small ε values. Recently, Chekuri and Khanna [4] improved on this result proposing a polynomial-time algorithm with worst-case performance ratio $1 + \varepsilon m + \mathrm{O}(\ln \varepsilon^{-1})$, for any $\varepsilon > 0$, which leads to a polynomial-time algorithm with worst-case performance ratio $\mathrm{O}(\ln m)$ when the dimension m is fixed. Also these algorithms seem to be only of theoretical interest. On the negative result side, Woeginger [30] showed that, even if $m = 2$, m-DVPP does not have an asymptotic polynomial time approximation scheme unless $\mathrm{P} = \mathrm{NP}$. We are not aware of any exact algorithm for m-DVPP in the literature.

This paper deals with the 2-DVPP. The problem was studied by Spieksma [25], who mentions applications in loading, scheduling, and layout design, considers lower bounding and heuristic procedures, and uses them within a branch-and-bound scheme, giving computational results for instances with up to 100 items. Han et al. [19] present heuristic and exact algorithms for a variant of 2-DVPP, where the bins are not identical.

Let n be the number of items, $N := \{1, \ldots, n\}$ represent the item set, and $M := \{1, \ldots, m\}$ represent the set of available bins, assuming that m is not smaller than the optimal solution value (e.g., $m = n$). Moreover, let us denote the weight of item j, $j \in N$, on the first and the second dimension by w_j and v_j, respectively, and the capacity of the bins on the first and the second dimension by c and d, respectively. Without loss of generality, we assume $c = d = 1$, and $w_j, v_j \leqslant 1$ for $j \in N$, unless explicitly stated.

A significant difference between BPP and 2-DVPP (and more generally m-DVPP) is that in BPP the items can be sorted according to their weights. This property allows one to develop dominance procedures which are of great help in reducing the size of a given BPP instance, and in pruning nodes in branch-and-bound algorithms (see [21]). These procedures cannot be adapted to 2-DVPP. Nevertheless, it is sometimes convenient to have a criterion for sorting the items of a 2-DVPP instance according to their "size". This issue is discussed throughout the paper.

The paper is organized as follows. Section 2 presents two alternative integer programming formulations for 2-DVPP, that will be used in the remainder of the paper. In Section 3 we describe several lower bounds, illustrating practical algorithms for their computation. Moreover, we determine an upper bound on the worst-case performance of a class of lower bounding procedures derived from BPP and prove that for all problem instances the lower bound associated with a huge linear programming relaxation is not smaller than all the other lower bounds we consider. In Sections 4 and 5 we deal with heuristic and exact algorithms, respectively. Finally, extensive computational experiments are reported in Section 6, showing that in some cases a combinatorial approach allows a fast solution of the problem, while in other cases one has to resort to a huge formulation for finding optimal solutions. Our results compare favorably with previous approaches to the problem.

2. Mathematical formulations

A possible Integer Linear Programming (ILP) model for 2-DVPP is the following:

$$\min \sum_{i \in M} y_i \tag{1}$$

subject to

$$\sum_{i \in M} x_{ij} = 1, \qquad j \in N, \tag{2}$$

$$\sum_{j \in N} w_j x_{ij} \leqslant y_i, \quad i \in M, \tag{3}$$

$$\sum_{j \in N} v_j x_{ij} \leqslant y_i, \quad i \in M, \tag{4}$$

$$0 \leqslant x_{ij} \leqslant y_i \leqslant 1, \quad i \in M, \; j \in N, \tag{5}$$

$$x_{ij}, y_i \text{ integer}, \qquad i \in M, \quad j \in N, \tag{6}$$

where $y_i = 1$ if and only if bin i is used and $x_{ij} = 1$ if and only if item j is packed into bin i.

The Linear Programming (LP) relaxation of this formulation, defined by (1)–(5), has the advantage of being solvable in O(n) time, as in [2]. On the other hand, the corresponding lower bound value is sometimes a long way from the optimum, as shown in the next section. Furthermore, the wide symmetry of the variables makes model (1)–(6) completely unusable for solving 2-DVPP instances with reasonable n values within a classical branch-and-bound algorithm based on LP relaxation.

An alternative ILP formulation of 2-DVPP, along the same line as the well-known Gilmore–Gomory formulation of the cutting stock problem [15,16], is the following. Let $\mathscr{S}$ be the family of all the *inclusion maximal* item sets that fit into a single bin, i.e., $\mathscr{S}:=\{S \subseteq N\colon \sum_{j\in S} w_j \leqslant 1,\ \sum_{j\in S} v_j \leqslant 1,\ \sum_{j\in S\cup\{i\}} w_j > 1$ or $\sum_{j\in S\cup\{i\}} v_j > 1$ for every $i \in N \setminus S\}$. A set covering formulation of 2-DVPP is

$$\min \sum_{S\in\mathscr{S}} \sigma_S \tag{7}$$

subject to

$$\sum_{S\ni j} \sigma_S \geqslant 1, \quad j \in N, \tag{8}$$

$$\sigma_S \geqslant 0, \quad S \in \mathscr{S}, \tag{9}$$

$$\sigma_S \text{ integer}, \quad S \in \mathscr{S}, \tag{10}$$

where σ_S is a binary variable taking the value 1 if and only if item set S is packed into a bin. (It is easy to show that both this formulation and its linear programming relaxation (7)–(9) are equivalent to their counterparts where $\mathscr{S}$ contains *all* the item sets that fit into a single bin, and in constraints (8) the $\geqslant$ is replaced by $=$; see [3].) The obvious disadvantage of this formulation is the possibly huge (exponential in n) number of variables: its LP relaxation (7)–(9), which is known to be NP-hard (see, e.g., [3]), can be solved by column-generation techniques (as discussed in the next section), but this may take quite a long time. On the other hand, even if the corresponding LP solution tends to be widely fractional, the rounded-up LP solution value is usually equal to the integer optimum.

The spirit of our approach to 2-DVPP is to try to solve the problem to optimality, within a short computing time, without explicitly using formulation (7)–(10). If we fail, i.e., the gap between the lower and upper bounds is not closed, we tackle the problem by first computing a tight lower bound by solving the LP relaxation (7)–(9), and then, if needed, by applying heuristic and exact algorithms based on this relaxation.

The dual of (7)–(9) reads

$$\max \sum_{j\in N} \pi_j \tag{11}$$

subject to

$$\sum_{j\in S} \pi_j \leqslant 1, \quad S \in \mathscr{S}, \tag{12}$$

$$\pi_j \geqslant 0, \quad j \in N. \tag{13}$$

Due to the tightness of the relaxation, component π_j^* in an optimal solution π^* gives reliable information about the "difficulty" of packing item j. Since we sometimes need a criterion for sorting the items, these values would be particularly suitable for this purpose. Nevertheless, as long as this LP relaxation is not involved in the solution of the problem, as discussed above, we sort the items according to decreasing values of $s_j := \lambda w_j + (1-\lambda)v_j$ ($j \in N$), where $\lambda := \sum_{j\in N} w_j / \sum_{j\in N}(w_j + v_j)$. Because of its definition, we call s_j the *surrogate weight* of item j.

3. Lower bounds

In the sequel, both a lower bounding procedure and the corresponding lower bound value are denoted by the same symbol. Moreover, given an item subset $S \subseteq N$, we let $L(S)$ denote lower bound L for the 2-DVPP instance corresponding to item set S. This section is organized as follows. First of all, we will give a description of the various lower bounds used. Then, in Section 3.1 we will describe effective algorithms for the computation of these lower bounds and briefly illustrate their performances in practice. Finally, in Section 3.2 we will study some theoretical properties of these bounds.

The first lower bound introduced by Spieksma [25], called L_C, is given by

$$L_C := \max\left\{ \left\lceil \sum_{j\in N} w_j \right\rceil, \left\lceil \sum_{j\in N} v_j \right\rceil \right\}.$$

A class of lower bounding procedures that can be derived from that proposed for BPP by Chen and Srivastava [5] is the following. For a given integer $d > 0$, find an inclusion maximal item set $S \subseteq N$ such that d is the maximum number of items in S that can be packed into a bin. Note that, whereas in the BPP case it is easy to find such a set S which is *maximum*, i.e., which has the largest possible cardinality, it is not clear how to find a maximum set S in the 2-DVPP case. Define the lower bound L_d as the optimal solution value of the 2-DVPP instance defined by the items in S only. Once S is given, for $d = 1$ and $d = 2$ L_d can be computed in polynomial time, while for $d \geqslant 3$ computation of L_d is NP-hard in the strong sense (as 3-partition can easily be reduced to it, see Garey and Johnson [13]).

The second lower bounding procedure proposed in [25] computes L_1 (i.e., L_d for $d = 1$) as described above, showing how to find an inclusion maximal subset S which is in fact *maximum*. In other words, [25] shows how to find the set S for which L_1 is maximum.

The first lower bounding procedure we propose computes L_2 (i.e., L_d for $d = 2$) as defined above. The set S is found heuristically in this case, as explained in

Section 3.1. We do not know if the determination of a set S such that the associated value of L_2 is maximum can be carried out efficiently.

The second new lower bounding procedure we propose, called L_H, was inspired by the Martello–Toth lower bound L_2 for BPP (see [22]). Given a partition of the items into 3 subsets N_1, N_2 and N_3, such that j_1 and j_2 cannot fit in the same bin for all $j_1 \in N_1$ and $j_2 \in N_2$, a valid lower bound is obtained as the sum of two lower bounds computed for the 2-DVPP instances with items in N_1 only, and items in N_2 only, respectively. We consider the following two possibilities for defining N_1, N_2, N_3. First, we let $N_1 := \{j \in N\colon w_j > 1 - \alpha \text{ and } v_j > 1 - \beta\}$, $N_2 := \{j \in N \setminus N_1\colon w_j \geqslant \alpha \text{ or } v_j \geqslant \beta\}$, and $N_3 := N \setminus (N_1 \cup N_2)$, where $\alpha, \beta \in [0, 1/2]$. Items in N_1 are clearly pairwise incompatible, therefore a valid lower bound is obtained as $L'(\alpha,\beta) := |N_1| + \max\{L_C(N_2), L_1(N_2), L_2(N_2)\}$. Also, we consider the partition defined by $N_1 := \{j \in N\colon w_j > 1 - \alpha \text{ or } v_j > 1 - \beta\}$, $N_2 := \{j \in N \setminus N_1\colon w_j \geqslant \alpha \text{ and } v_j \geqslant \beta\}$, and $N_3 := N \setminus (N_1 \cup N_2)$, where again $\alpha, \beta \in [0, 1/2]$. Since at most two items in N_1 can be packed into the same bin, a corresponding lower bound is given by $L''(\alpha,\beta) := L_2(N_1) + \max\{L_C(N_2), L_1(N_2), L_2(N_2)\}$. We then define lower bound L_H as $\max\{\max_{\alpha,\beta\in[0,1/2]} L'(\alpha,\beta), \max_{\alpha,\beta\in[0,1/2]} L''(\alpha,\beta)\}$.

The last lower bound we use, called L_B, is defined as the rounded-up value of the optimal solution to the LP relaxation (7)–(9).

3.1. Computation of the lower bounds

Lower bound L_C is trivially computed in $O(n)$ time, and is the maximum between the continuous lower-bound values of the two BPPs obtained by neglecting the second and the first dimension, respectively (see [22]). In fact, in a more general setting Caprara [2] proved that L_C is the rounded-up value of the LP relaxation (1)–(5).

Theorem 1 (Caprara [2]). *The LP relaxation* (1)–(5) *can be solved in* $O(n)$ *time, and its optimal solution value* z_C^* *is such that* $\lceil z_C^* \rceil = L_C$.

In the sequel, we will often refer to the so-called *compatibility graph* associated with 2-DVPP, which is defined as follows. Two items $j, k \in N$ are called *compatible* if $w_j + w_k \leqslant 1$ and $v_j + v_k \leqslant 1$, i.e., if j and k can be packed into the same bin, and *incompatible* otherwise. The compatibility graph $G = (N, E)$ is the undirected graph having a node corresponding to each item, and an edge $(j,k) \in E$ for each pair (j,k) of compatible items. A *stable set* of G is a node set $S \subseteq N$ such that $(i,j) \notin E$ for all $i,j \in S$. Such a set corresponds to a set of items that must be packed into different bins. Analogously, a *clique* of G is a node set $S \subseteq N$ such that $(i,j) \in E$ for all $i,j \in S$. Let $\alpha(G)$ denote the maximum cardinality of a stable set of G. A *partition into cliques* of G is a partition of the node set V into node sets $K_1, \ldots, K_q$ such that K_i is a clique for $i = 1, \ldots, q$ (where q is the cardinality of the partition). Let $\chi(G)$ denote the minimum cardinality of a partition into cliques of G. A *subgraph* of G defined by $S \subset N$ is the graph $G(S) = (S, E(S))$, where $E(S) := \{(i,j) \in E\colon i,j \in S\}$. A graph $G=(N,E)$ is called *perfect* if $\alpha(G(S))=\chi(G(S))$ for all $S\subseteq V$, i.e., in G and in all its subgraphs a minimum-cardinality partition into cliques has the same value as a maximum-cardinality stable set. Hammer and Mahadev [18] proved the following result.

Theorem 2 (Hammer and Mahadev [18]). *The compatibility graph G associated with 2-DVPP is* perfect.

Hammer and Mahadev [18] show how to compute a minimum-cardinality partition into cliques of G in $O(n^2)$ time. By Theorem 2, this computation also yields the maximum cardinality of a stable set of G, i.e., the maximum number p of pairwise incompatible items. Hence, the procedure proposed in [18] can be used to compute L_1 (see [25]), set equal to p. Note that the procedure avoids the explicit determination of S and at the same time finds the best possible value of L_1 over all item sets.

We now illustrate an $O(n^2)$ algorithm for the computation of L_2 once S is given. Clearly, if the maximum number of items in N that fit into a bin is two, then an optimal solution to 2-DVPP can be efficiently determined by finding a *maximum-cardinality* (*maximum*, for short) *matching* of G. More precisely, let $M\subseteq E$ be a matching of G, i.e., for each $i\in N$ there exists at most one edge in M incident to i. For each edge $(i,j)\in M$, nodes i and j are called *matched*, and i,j is called a *matched pair*. A feasible solution to 2-DVPP is then obtained by using one bin for packing the items corresponding to each matched pair, and one bin for each item whose corresponding node is not matched, the number of bins being $n-|M|$. We then show a simple algorithm for computing a maximum matching of G in $O(n^2)$ time. The proof of correctness of the algorithm is based on the following lemmas.

Lemma 1. *Given a graph $G=(N,E)$ and a node $i\in N$ having at least one incident edge, there always exists a maximum matching of G where i is matched.*

Proof. Consider a maximum matching M of G, and suppose no edge in M is incident to i. Let $e'=(i,j)$ be any edge incident to i, and let $e=(k,j)$ be the edge in M incident to j (if none exists M is not maximum). $M':=M\setminus\{e\}\cup\{e'\}$ is a maximum matching where i is matched. □

Lemma 2. *Given the compatibility graph $G=(N,E)$ for 2-DVPP, let i be such that $w_i=\max_{k\in N}w_k$ and suppose there exist items compatible with i. Let j be such that $v_j=\max\{v_k\colon (i,k)\in E\}$. There then exists a maximum matching of G containing edge (i,j).*

Proof. Consider a maximum matching M of G where i is matched (such a matching exists from Lemma 1) and suppose $(i,j)\notin M$. Let (i,k) be the edge in M incident to i. If j is not matched, define a new maximum matching $M':=M\setminus\{(i,k)\}\cup\{(i,j)\}$. Otherwise, let (j,l) be the edge in M incident to j. We claim that $(k,l)\in E$, i.e., objects k and l are compatible. Indeed $w_k + w_l \leqslant w_k + w_i \leqslant 1$ since $(i,k) \in E$, and $v_k + v_l \leqslant v_j + v_l \leqslant 1$ since $(j,l) \in E$. Therefore $M' := M \setminus \{(i,k),(j,l)\} \cup \{(i,j),(k,l)\}$ is a maximum matching of G. □

By iteratively applying Lemma 2, one can easily compute a maximum matching of G. The resulting procedure is described below.

```
procedure 2-DVPP MATCHING;
input: the compatibility graph $G$ associated with an instance $(n,(w_j,v_j), j \in N)$
        of 2-DVPP;
output: a maximum matching $M$ of $G$;
begin
  $M := \emptyset$; $S := N$;
  while $S \neq \emptyset$ do
    $i := \arg\max_{k \in S} w_k$;
    $C := \{k \in S \setminus \{i\}: w_k + w_i \leqslant 1$ and $v_k + v_i \leqslant 1\}$;
    if $C \neq \emptyset$ then
      $j := \arg\max_{k \in C} v_k$;
      $S := S \setminus \{i,j\}$; $M := M \cup \{(i,j)\}$
    else
      $S := S \setminus \{i\}$
    end
  end
end.
```

This algorithm is very similar to that used in [25] for computing a maximum stable set of G, having time complexity $O(n^2)$. Given an instance of 2-DVPP, we then compute the lower bound L_2 by heuristically determining an inclusion maximal subset S of items such that no more than two items in S can be contained in the same bin, and by solving to optimality, through the above procedure, the 2-DVPP instance corresponding to S. More precisely, we initialize $S := \{j \in N: w_j > 1/3$ and $v_j > 1/3\}$. We then consider the items in $N \setminus S$ in decreasing order of surrogate weights (as defined in Section 2), and for each of them, say i, we let $S := S \cup \{i\}$ if $w_i + w_j + w_k > 1$ or $v_i + v_j + v_k > 1$ for each pair j,k of items in S. This test can be carried out in $O(|S| \log |S|)$ time through procedure CHECK INCOMPATIBILITY in the Appendix, modified so as to consider bin capacities $1 - w_i$ and $1 - v_i$, respectively. Hence the overall definition of S requires $O(n^2 \log n)$ time in the worst case, but in practice it turns out to be faster than the application of procedure 2-DVPP MATCHING.

For the computation of L_H, it is easy to check that only α, β pairs such that $\alpha = \min\{w_i, 1 - w_i\}, \beta = \min\{v_j, 1 - v_j\}$ for some $i,j \in N$ are worth considering. Therefore, L_H is determined by considering all the $O(n^2)$ significant α, β pairs. The overall computational complexity is $O(n^4 \log n)$ in the worst case, even if in practice the running time turns out to be not so bad. In our implementation, on input we give the value of the best heuristic solution, say z_H, to the procedure which computes L_H: the procedure is stopped if for some values of α and β, either $L'(\alpha,\beta) = z_H$ or $L''(\alpha,\beta) = z_H$.

The solution of the LP relaxation (7)–(9) in the computation of L_B is carried out in a standard way by column generation, following the original scheme by Gilmore and Gomory [15,16]. More precisely, we initialize the LP relaxation with a subset $\mathscr{C} \subset \mathscr{S}$ of the variables, each corresponding to an item subset packed into a bin by a heuristic solution to the problem. We then iteratively (i) solve the current LP relaxation; (ii) check whether there exists a variable σ_S, $S \in \mathscr{S} \setminus \mathscr{C}$, with negative reduced cost with respect to the current optimal dual solution; (iii) add σ_S to the LP; and repeat until no negative reduced-cost variable exists. According to (11)–(13), the reduced cost of variable σ_S with respect to the current dual solution π^* is given by $1 - \sum_{j \in S} \pi_j^*$. Therefore, in order to find, if any, a variable which has negative reduced cost, one can solve the 2-*Constraint Knapsack Problem* (2-CKP)

$$\max \sum_{j \in N} \pi_j^* \eta_j \tag{14}$$

subject to

$$\sum_{j \in N} w_j \eta_j \leqslant 1, \tag{15}$$

$$\sum_{j \in N} v_j \eta_j \leqslant 1, \tag{16}$$

$$\eta_j \in \{0,1\}, \quad j \in N. \tag{17}$$

Let $\bar{\eta}$ be an optimal solution, and $\bar{\rho}$ be its value. If $\bar{\rho} \leqslant 1$, then no negative reduced-cost variable exists, otherwise item set $S := \{j \in N: \bar{\eta}_j = 1\}$ corresponds to the variable σ_S having the negative reduced cost with the largest absolute value. Since the above 2-CKP is solvable in pseudo-polynomial time by dynamic programming, the overall LP relaxation (7)–(9) is solvable in pseudo-polynomial time, see Grötschel et al. [17].

3.2. Theoretical properties of the lower bounds

The *worst-case performance ratio* of lower bound L, WCPR(L), is defined as the infimum of L/z_{OPT} over all the 2-DVPP instances, where z_{OPT} denotes the optimal solution value to the problem, see [21].

L_C behaves well experimentally for instances with "small" items, i.e., items whose weights are small in relation to bin capacities. For BPP it is known that WCPR(L_C) = 1/2 (see, e.g., [22]). In [2], the following properties are shown.

Theorem 3 (Caprara [2]). *For all 2-DVPP instances with $w_j, v_j \leqslant 1/k$, $k \geqslant 1$ and integer*, WCPR$(L_C) \geqslant 1/(1+2/k)$.

As a corollary, in the general case, i.e., for $k = 1$, we have

Corollary 1 (Caprara [2]). $\mathrm{WCPR}(L_C) = 1/3$.

Consider a given integer $d > 0$. Clearly, $\mathrm{WCPR}(L_d) = 0$, since $|S| \leqslant d$ and $L_d = 1$ for all instances for which $w_j, v_j \leqslant 1/(d+1)$, $j \in N$. In the BPP case, Chen and Srivastava prove in [5] that $\mathrm{WCPR}(\max\{L_C, L_d\}) = 2/3$. The following theorem gives an upper bound on $\mathrm{WCPR}(\max\{L_C, L_d\})$ for our problem.

Theorem 4. *For every positive integer* d, $1/3 \leqslant \mathrm{WCPR}(\max\{L_C, L_d\}) \leqslant (d+1)/(2d+3)$.

Proof. The first inequality clearly follows from Corollary 1. Consider now the class of instances with $n = (d+3)s$, $w_j = 1/(d+2) + \varepsilon$, $v_j = 2\varepsilon$ for $j = 1, \ldots, (d+2)s$; $w_j = 0$, $v_j = 1 - \varepsilon$ for $j = (d+2)s + 1, \ldots, n$, where s is an integer number and $\varepsilon \leqslant 1/s(2d+3)$. For these instances, $z_{\mathrm{OPT}} = \lceil s(2d+3)/(d+1) \rceil \geqslant s(2d+3)/(d+1)$ and $L_C = L_d = s+1$, therefore $\max\{L_C, L_d\}/z_{\mathrm{OPT}} \leqslant (s+1)(d+1)/s(2d+3)$, which converges to $(d+1)/(2d+3)$ as s tends to infinity. □

Therefore for no value of d is the lower bound obtained better than 1/2 the optimum in the worst case. We now turn our attention to the values of d for which the associated lower bound can be computed efficiently.

Clearly, L_1 can be a good lower bound only for instances with some "big" items, i.e., items whose weight, for at least one dimension, is greater than half the bin capacity. From the properties of bound L_d it can be seen immediately that $\mathrm{WCPR}(L_1) = 0$, whereas from Theorem 4 one gets

Corollary 2. $1/3 \leqslant \mathrm{WCPR}(\max\{L_C, L_1\}) \leqslant 2/5$.

As to L_2, from the general properties of lower bound L_d, $\mathrm{WCPR}(L_2) = 0$ and, from Theorem 4,

Corollary 3. $1/3 \leqslant \mathrm{WCPR}(\max\{L_C, L_1, L_2\}) \leqslant 3/7$.

By considering the class of instances in the proof of Theorem 4 with $d = 1$, one gets $L_C = L_1 = s + 1$, $L_2 = z_{\mathrm{OPT}} = \lceil 5s/2 \rceil$, showing that L_2 can be better than L_1 and L_C. Viceversa, if S is constructed in the "wrong" way, bound L_2 can in principle be considerably worse than L_1. For example, consider the class of instances with $n = 2s + 2$, $w_1 = w_2 = \frac{1}{6}$, $v_1 = v_2 = \frac{1}{2}$; $w_j = 0$, $v_j = 1$ for $j = 3, \ldots, s+2$; $w_j = \frac{2}{3}, v_j = 0$ for $j = s+3, \ldots, 2s+2$. The value for parameter λ used in the definition of the surrogate weights is 1/4 (see Section 2). Accordingly, S is constructed by inserting in turn items $3, \ldots, s+2$, and then 1 and 2. The corresponding L_2 value is $s+1$, while the optimal solution value is clearly $2s$, equal to bound L_1. Nevertheless, in our computational experiments bound L_2 outperformed bound L_1, yielding a better or equal value for almost all the instances tried, with a comparable computational effort.

We could not find any significant upper bound on the worst-case performance ratio of lower bounds L_H and L_B. Instead, we compare these bounds to each other as well as to the previous ones.

In the computation of L_H, when α and β assume their minimum values, $N_1 = \emptyset$, hence $L_H \geqslant \max\{L_C, L_1, L_2\}$. For the family of instances of Theorem 4, $L_H = L''(\alpha,\beta) = 2s+1$ for every $\alpha \in [0, 1/(d+2)+\varepsilon]$, $\beta \in (\varepsilon, 2\varepsilon]$. Therefore, for these instances L_H is consistently better than the previous bounds. In our computational experiments, lower bound L_H sometimes turns out to be better than all the others, and allows one to prove that the heuristic solution in hand is optimal.

As mentioned in Section 2, the computation of L_B requires a considerable amount of time if compared with the computation of the other lower bounds (including L_H), but, in practice, the corresponding value turns out to be equal to the optimum basically in all cases. We conclude the section by showing that lower bound L_B dominates all the others, i.e., it is never smaller.

We start by proving two results about the structure of the incompatibility graph G. The first one is a simple corollary of Theorem 2. An *odd hole* of G is a cycle C formed by k edges, $k \geqslant 5$ and odd, such that there is no edge in $E \setminus C$ joining two nodes visited by C.

Lemma 3. *G contains no odd hole.*

A $\bar{K}_{3,3}$ is a graph of 6 nodes corresponding to two node-disjoint triangles, i.e., a graph with node set $\{j_1, j_2, j_3, j_4, j_5, j_6\}$ and edge set $\{(j_1,j_2), (j_2,j_3), (j_3,j_1), (j_4,j_5), (j_5,j_6), (j_6,j_4)\}$.

Lemma 4. *G contains no* $\bar{K}_{3,3}$.

Proof. Suppose the claim is false, and let $\{j_1, j_2, j_3, j_4, j_5, j_6\}$ and $\{(j_1,j_2), (j_2,j_3), (j_3,j_1), (j_4,j_5), (j_5,j_6), (j_6,j_4)\}$ be the node set and the edge set of a $\bar{K}_{3,3}$ of G, respectively. One can suppose without loss of generality that item pairs j_1, j_4 and j_2, j_5 are incompatible because of the first dimension, i.e., $w_{j_1} + w_{j_4} > 1$ and $w_{j_2} + w_{j_5} > 1$, and also that $w_{j_1} = \max\{w_{j_1}, w_{j_2}, w_{j_4}, w_{j_5}\}$. But then $w_{j_1} + w_{j_2} > 1$, contradicting the fact that $(j_1, j_2) \in E$. $\square$

The following lemma shows that, in the computation of lower bound L_H, given a partition of the items into subsets N_1, N_2 and N_3 such that j_1 and j_2 are incompatible for all $j_1 \in N_1$ and $j_2 \in N_2$, bound L_1 yields the optimal solution value for one of the two instances defined by item sets N_1 and N_2, respectively.

Lemma 5. *Given two 2-DVPP instances defined by item sets N_1 and N_2, such that for each pair $j_1 \in N_1$, $j_2 \in N_2$, j_1 and j_2 are incompatible, the computation of lower bound L_1 yields the optimal solution value for at least one of the two instances.*

Proof. Consider the compatibility graph $G=(N_1 \cup N_2, E)$ associated with the instance defined by item set $N_1 \cup N_2$, and let $G(N_1)$ and $G(N_2)$ be the subgraphs of G induced by item sets N_1 and N_2, respectively. If either $|N_1| \leqslant 2$ or $|N_2| \leqslant 2$ the theorem is clearly true, so we assume in the following $|N_1| \geqslant 3$ and $|N_2| \geqslant 3$.

We first claim that either $G(N_1)$ or $G(N_2)$ contains no triangle, i.e., no clique of size 3. Indeed, if this was false, since there is no edge connecting a node in N_1 with a node in N_2, G would contain a $\bar{K}_{3,3}$, which is not possible by Lemma 4.

So suppose, without loss of generality, that $G(N_1)$ contains no triangle. In this case, the optimal solution value to the associated 2-DVPP instance is given by $L_1(N_1)$ (recall that $L(S)$ denotes lower bound L for the instance defined by item set S). Indeed, the optimal solution value is given by $L_2(N_1)$, as at most 2 items can be packed into a single bin. Since the maximum size of a clique of $G(N_1)$ is 2, $L_2(N_1)$ also gives the minimum number of cliques into which node set N_1 can be partitioned. As $G(N_1)$ is perfect (see Theorem 2), this number coincides with the maximum cardinality of a stable set of $G(N_1)$, i.e., with $L_1(N_1)$. $\square$

We are now ready to prove that lower bound L_B dominates L_H, and therefore also all the other lower bounds we considered.

Theorem 5. $L_B \geqslant L_H$

Proof. Remembering that $L_H \geqslant \max\{L_C, L_1, L_2\}$, we prove initially that $L_B \geqslant \max\{L_C, L_1, L_2\}$.

First of all, we show feasible dual solutions to (11)–(13) whose rounded-up values are equal to L_C and L_1, respectively.

For the L_C case, both solutions $\pi_j^* := w_j$ for $j \in N$ and $\pi_j^* := v_j$ for $j \in N$ are clearly feasible, and L_C coincides with the rounded-up value of one of them.

For the L_1 case, let $S \subseteq N$ correspond to a maximum stable set of the compatibility graph G. The solution $\pi_j^* := 1$ for $j \in S$, $\pi_j^* := 0$ for $j \in N \setminus S$ is feasible and has value $|S| = L_1$.

The proof for L_2 is much more involved. The procedure that computes L_2 solves to optimality a 2-DVPP subinstance of the original instance where at most two items fit into a bin. Let G be the compatibility graph associated with this subinstance. If G has isolated nodes, i.e., items exist incompatible with all the other items, it is easy to see that the family $\mathscr{S}$ contains inclusion maximal subsets of one item only, and the corresponding optimal values of the variables in both (7)–(10) and its LP relaxation take the value 1. These nodes, and the corresponding variables, can be removed, since they give the same contribution to both L_B and L_2. Hence we suppose that the compatibility graph G contains no isolated node. In this case, the family $\mathscr{S}$ of inclusion maximal item sets that fit into a bin coincides with E, hence formulation (7)–(10) has the form

$$\min \sum_{e \in E} \sigma_e \tag{18}$$

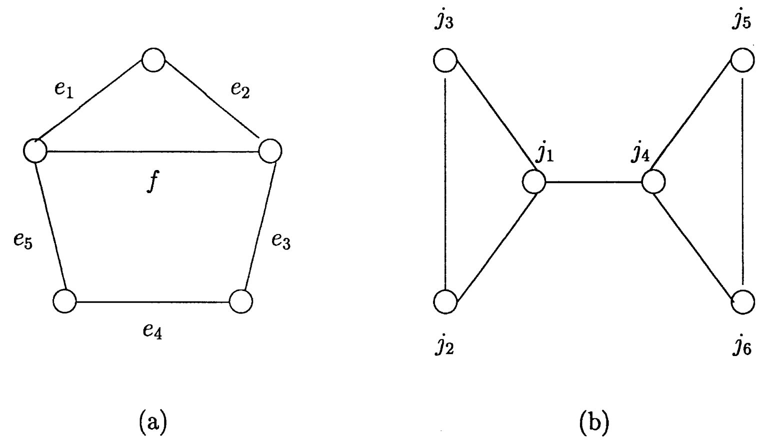

Fig. 1. Illustration of some steps in the proof of Theorem 5.

subject to

$$\sum_{e \in \delta(j)} \sigma_e \geqslant 1, \quad j \in N, \tag{19}$$

$$\sigma_e \geqslant 0, \quad e \in E, \tag{20}$$

$$\sigma_e \text{ integer}, \quad e \in E, \tag{21}$$

where $\delta(j)$ denotes the set of edges in E incident to j. We show that the optimal value of the LP relaxation (18)–(20) is within 1/2 of the optimal value of (18)–(21), and therefore that $L_B = L_2 = z_{\text{OPT}}$ for this subinstance.

For any graph G, it has been shown (see, e.g., [14]) that there exists an optimal solution σ^* to (18)–(20) such that

(i) each fractional entry of σ^* is equal to 1/2;
(ii) the connected components of the subgraph G^* of G obtained by removing from G all the edges in $e \in E$ such that σ^*_e is integer, are either isolated nodes or odd cycles of G.

We first show that each connected component of G^* which is not an isolated node can be assumed to be a triangle (i.e., an odd cycle of 3 edges) without loss of generality. Consider a cycle $C = \{e_1, e_2, \ldots, e_k\}$ of G^*, where k is odd, and suppose $k \geqslant 5$: from Lemma 3 it follows (see Fig. 1(a)) that there exists an edge $f \in E \setminus C$ such that $C \cup \{f\}$ defines a triangle T and an even cycle D of G having f as the only common edge. (Note that $\sigma^*_f = 0$.) One can assume edges in C have been numbered so as to have $T = \{e_1, e_2, f\}$ and $D = \{f, e_3, e_4, \ldots, e_k\}$. By redefining $\sigma^*_e := 1/2$ for $e \in T$; $\sigma^*_e := 1$ for $e = e_i$, $4 \leqslant i \leqslant k-1$, i even; $\sigma^*_e := 0$ for $e = e_i$, $3 \leqslant i \leqslant k$, i odd, and leaving the other components of σ^* unchanged, one gets a new solution σ^* to (18)–(20) with the same value as the original one.

One can also assume without loss of generality that no pair of triangles T_1, T_2 in G^* is such that there exists an edge in E joining a node in T_1 to a node in T_2 (see

Fig. 1(b)). Indeed, let $T_1 := \{(j_1,j_2),(j_2,j_3),(j_3,j_1)\}$ and $T_2 := \{(j_4,j_5),(j_5,j_6),(j_6,j_4)\}$, and suppose $(j_1,j_4) \in E$. Redefining $\sigma_{j_1j_4} := 1$, $\sigma_{j_2,j_3} := 1$, $\sigma_{j_5,j_6} := 1$, $\sigma_{j_1,j_2} := 0$, $\sigma_{j_3,j_1} := 0$, $\sigma_{j_4,j_5} := 0$, $\sigma_{j_6,j_4} := 0$, and leaving the other components of σ^* unchanged, one gets a new solution σ^* to (18)–(20) with the same value as the original one and such that neither T_1 nor T_2 is a connected component of the new G^*; namely the two connected components T_1 and T_2 have been replaced by the three connected components $\{j_1,j_4\}$, $\{j_2,j_3\}$ and $\{j_5,j_6\}$.

Finally, by the two properties shown above and by Lemma 4, one can assume that G^* either contains no triangle, in which case σ^* is integer and the proof is complete, or contains exactly one triangle $T = \{e_1,e_2,e_3\}$ such that $\sigma^*_{e_1} = \sigma^*_{e_2} = \sigma^*_{e_3} = 1/2$. The integer solution σ^I defined by $\sigma^I_e := \sigma^*_e$ if $e \in E \setminus T$ and $\sigma^I_{e_1} := 1$, $\sigma^I_{e_2} := 1$, $\sigma^I_{e_3} := 0$ is feasible for (18)–(21) and its value is greater than that of σ^* by 1/2.

We complete the proof by showing that $L_B \geqslant L_H$. Remember that L_H is defined as the sum of two lower bounds, chosen from among L_C, L_1, L_2, computed on two item sets N_1 and N_2 such that no item in N_1 is compatible with any item of N_2 (and vice versa). Observe that one can construct a feasible dual solution π^* to (11)–(13) for the instance defined by items in $N_1 \cup N_2$ by considering dual solutions π^1 and π^2 to the instances defined by N_1 and N_2, respectively, and by setting $\pi^*_j := \pi^1_j$ if $j \in N_1$ and $\pi^*_j := \pi^2_j$ if $j \in N_2$. By Lemma 5, one can assume that the lower bound for N_1 used in the definition of L_H is $L_1(N_1)$, and π^1 is defined so as to have $L_1(N_1) = \lceil \sum_{j \in N_1} \pi^1_j \rceil = \sum_{j \in N_1} \pi^1_j$ (see above). Moreover, it is possible to define π^2 so as to have $\lceil \sum_{j \in N_2} \pi^2_j \rceil \geqslant \max\{L_C(N_2), L_1(N_2), L_2(N_2)\}$ (see above). Therefore, $L_B = \lceil \sum_{j \in N_1} \pi^1_j + \sum_{j \in N_2} \pi^2_j \rceil = L_1(N_1) + \lceil \sum_{j \in N_2} \pi^2_j \rceil \geqslant L_1(N_1) + \max\{L_C(N_2), L_1(N_2), L_2(N_2)\} \geqslant L_H$. $\square$

4. Heuristic algorithms

The first heuristic algorithms for 2-DVPP are a natural adaptation of the greedy heuristics for BPP. Among these, the most effective ones consider the items sorted according to decreasing weights. As mentioned in the introduction, it is not clear how to sort the items according to their "dimension" in the 2-DVPP case, therefore different reasonable criteria are worth considering.

The most popular greedy heuristic for BPP is First Fit Decreasing (FFD). FFD considers the items in decreasing order of weights; each item j is packed into the first non-empty bin into which it fits, if no such bin exists a new bin is initialized to contain j. In [12] an adaptation of FFD to the 2-DVPP case is proposed, where the items are considered in decreasing order of $\max\{w_j, v_j\}$ values; the corresponding heuristic, called here 2FFD, can be implemented to run in $O(n \log n)$ time. In the iterative heuristic proposed in [25], here denoted by $2FFD_\mu$, at each iteration an adaptation of FFD is applied, which considers the items in decreasing order of $\mu w_j + v_j$ values, where μ is a non-negative parameter. In the first iteration, the value of μ (say $\mu = 1$) is given on input, and all the items are packed. At the end of each iteration, the solution is examined to decide which bins are "well-filled": the corresponding items are removed from the problem, and μ is updated (see [25,24]). This heuristic requires $O(n^2 \log n)$ time. Another possible ranking we consider is the one given by the surrogate weights, as defined in Section 2. The corresponding heuristic is called 2FFD$_\lambda$, and requires $O(n \log n)$ time. We also consider the above adaptations to the famous Best Fit Decreasing (BFD) heuristic for BPP, that iteratively packs each item into the bin with minimal residual capacity into which it fits. This leads to algorithms 2BFD, 2BFD$_\mu$ and 2BFD$_\lambda$, with time complexities $O(n \log n)$, $O(n^2 \log n)$ and $O(n \log n)$, respectively.

We apply a simple exchange procedure to try to improve each of the heuristic solutions returned by the procedures above. This procedure, called 2REF, iteratively executes the following step.

The bin b in the current solution containing a set of items with minimum overall surrogate weight is considered. From this bin, we pick the item j with maximum surrogate weight s_j, and pack j into another non-empty bin. We choose the bin where j can be packed by picking out a set S (possibly $S = \emptyset$) of items with minimum overall surrogate weight $\sum_{i \in S} s_i$, breaking ties in favor of the set with the largest cardinality. The items in S are then packed according to a BFD policy with respect to the surrogate weights. The step is complete when all the items in S have been packed, the complexity of each step being $O(n \log n)$.

Notice that the current solution after the application of a step might be better or worse than the one available before. In order to prevent cycling, we keep track of the exchanges made in one step so as to avoid doing the reverse move in the following steps, according to a *tabu search* policy. Extensive computational experiments have suggested stopping the procedure after 250 steps, since we observed that afterwards no improvement typically occurs.

Computational experience shows that the above-mentioned greedy heuristics followed by the exchange procedure, together with lower bounds L_C, L_1, L_2, L_H, often allow one to solve a 2-DVPP instance to proven optimality.

We now describe a more complex and time consuming heuristic, called H_M, mainly intended to produce good solutions for the instances where the number of items packed into a bin is typically 2 or 3. We consider the compatibility graph $G = (N, E)$ as defined in the previous section. For each edge $(i, j) \in E$ let $K_{ij} := \{k \in N \setminus \{i, j\}$: $w_i + w_j + w_k \leqslant 1$ and $v_i + v_j + v_k \leqslant 1\}$. Edge (i, j) is assigned the profit $p_{ij} := s_i + s_j + \alpha \max_{k \in K_{ij}} s_k$ (with $\max \emptyset := 0$), where α is a suitably defined non-negative parameter, set to 0.5 in our implementation.

Iteratively, heuristic H_M computes a maximum-profit matching of G, considers the edge (i, j) in the matching with highest profit, and packs into a single bin items i, j, and $k := \arg\max_{k \in K_{ij}} s_k$ if $K_{ij} \neq \emptyset$. The heuristic then tries to pack some of the remaining items into the bin, considering the items with larger surrogate weight first. The packed items are removed from the problem, and the procedure is iterated until a feasible overall solution is found. The overall complexity of H_M is $O(n^4)$, since each computation of the profit matrix and each solution of a matching require $O(n^3)$ time in the worst case. After the first step, each recomputation of the profit matrix is done parametrically. In practice, the running time is much better than in the worst case.

The last heuristic algorithm we propose, H_B, is based on the solution of the LP relaxation (7)–(9). It is a classical *diving* heuristic, that iteratively solves the LP relaxation and fixes the variable σ_S with highest value in the LP solution to 1, until a feasible solution is found. The LP solutions are clearly computed parametrically. We observed that column generation is worth doing also after fixing, since some variables with value 0, and therefore possibly not generated, in the first LP are often needed to get very good solutions. The complexity of this heuristic is pseudo-polynomial, and in practice a significant fraction of the computing time is spent for the first LP relaxation (7)–(9). H_B is far more time consuming than the other heuristics, but in most cases it finds better solutions.

5. Exact algorithms

The enumerative scheme proposed by Spieksma [25] works as follows. One bin at a time is considered, and items are put in it (following a certain order) until no further item can be packed. Then a lower bound for the reduced instance is computed: if the incumbent solution cannot be improved, a backtracking is performed, otherwise a new (empty) bin is considered. The aim is finding a feasible solution of value $k-1$, where k is the best solution value found so far. If any is found, k is updated and the procedure is restarted. A simple dominance test is also applied.

In our implementation we initially tested two different branch-and-bound algorithms for 2-DVPP. The spirit of these algorithms is very similar to the one of the branch-and-bound algorithm for BPP proposed by Martello and Toth [22]. Therefore, the reader is referred to [22] for further details, as well as examples of how branch-and-bound works on small instances. In the first one, called BB_1, we generate son-nodes using the "item-by-item" branching scheme, considering an item not yet packed and putting it, respectively, in each non-empty bin into which it fits, and in a new (empty) bin. At each node of the branch-decision tree, we first compute the lower bound $l+\max\{L_C(R),L_1(R),L_2(R)\}$, where l is the number of bins that cannot contain any further item, and R is the instance composed by the items which have not been packed yet, and by "aggregate" items. For each used bin h which can contain some further item, an aggregate item $n+h$ is defined, with $w_{n+h}:=\sum_{j\in B_h} w_j$ and $v_{n+h}:=\sum_{j\in B_h} v_j$, where B_h is the set of items packed into bin h. Then we try to improve the incumbent solution by applying the six greedy algorithms 2FFD, $2\mathrm{FFD}_\mu$, $2\mathrm{FFD}_\lambda$, 2BFD, $2\mathrm{BFD}_\mu$, and $2\mathrm{BFD}_\lambda$, described in the previous section. In the second algorithm, called BB_2, for each node of the tree we have a "current" bin. Son nodes are generated using the "bin-by-bin" branching scheme (similar to that proposed by Spieksma [25]), putting, in turn, each unassigned item in this bin. At the root node, and when the current bin cannot contain any further items, we compute the lower bound $l+\max\{L_C(R),L_1(R),L_2(R)\}$, where l is the number of bins used and R is the instance composed by the items which have not been packed, and we apply the greedy algorithms. In both algorithms, we use a depth-first search scheme. Moreover, at the root node heuristic

H_M is also applied, and each heuristic algorithm is followed by procedure 2REF.

The behavior of these two algorithms is not quite satisfactory, since they can solve some instances quickly, but often require a very long computing time to terminate. As for BPP, the initial gap between the lower and upper bound values is typically very small (1 in many cases), but a huge number of branch decision nodes is required for closing this gap. Sometimes, the initial lower bound is equal to the optimum, and all the effort is used in finding an optimal solution. A main problem of the above-described branch-and-bound algorithms is that they do not try to explore the "most promising" solutions first, but instead try to fill up the bins quite "blindly", exploring (at least partially) a number of "bad" solutions. This observation was the main inspiration for the development of a new branch-and-bound algorithm, described in the sequel.

The new branch-and-bound algorithm, called BB$_3$, is intended to explore first the solutions where the bins are filled-up as much as possible, with respect to the surrogate weights. More precisely, at each branch decision node, say $\mathscr{N}$, we have a set $R \subseteq N$ of items still to be packed ($R = N$ at the root node). We compute the lower bound $l + \max\{L_C(R), L_1(R), L_2(R)\}$, where l is the level of $\mathscr{N}$ in the branch decision tree (see below), and apply the greedy algorithms. If the node is not fathomed, we branch ideally by considering all the possible inclusion maximal subsets $S \subseteq R$ that fit into a bin. For each subset S, we pack the items in S into a bin and consider the corresponding subproblem, where $R{:=}R \setminus S$. Among these subproblems, those for which $\sum_{j\in S} s_j$ is higher are explored first. The actual implementation of this strategy works as follows. When node $\mathscr{N}$ is explored, we solve the 2-CKP

$$\max \sum_{j\in R} s_j \eta_j \tag{22}$$

subject to

$$\sum_{j\in R} w_j \eta_j \leqslant 1, \tag{23}$$

$$\sum_{j\in R} v_j \eta_j \leqslant 1, \tag{24}$$

$$\eta_j \in \{0,1\}, \quad j \in R, \tag{25}$$

yielding a solution η^1. S is defined as $S^1{:=}\{j \in R: \eta_j^1 = 1\}$, and a first subproblem is generated and explored, in a depth-first way, with $R{:=}R \setminus S^1$. When backtracking goes back to node $\mathscr{N}$, we solve the 2-CKP (22)–(25) with the further constraint

$$\sum_{j\in S^1} \eta_j \leqslant |S^1| - 1, \tag{26}$$

forbidding solution η^1, and no other solution. Given its solution η^2, the new subset $S^2{:=}\{j \in R: \eta_j^2 = 1\}$ is defined, and a second subproblem is generated and explored, with $R := R \setminus S^2$. This process is iterated, so that the pth backtracking to node $\mathcal{N}$ requires solving the 2-CKP (22)–(25) augmented by the constraints

$$\sum_{j \in S^k} \eta_j \leqslant |S^k| - 1, \quad k = 1, \ldots, p. \tag{27}$$

Clearly, such an approach can be effective only if node $\mathcal{N}$ is fathomed after a limited number of backtrackings. To this end, we apply a suitably defined dominance test.

Let z_H be the value of the best 2-DVPP solution so far. Let $z_T := z_H - 1$ be the maximum value of a target better solution (if any). Without loss of generality, we require the bin set up at the first level, say bin 1, to be such that $\sum_{j \in B_1} s_j \geqslant \sum_{j \in B_i} s_j$ for each bin i in the solution, where B_i is the set of items packed into bin i. This implies that

$$\sum_{j \in B_1} s_j \geqslant \frac{\sum_{j \in N} s_j}{z_T} \tag{28}$$

in any solution of value at most z_T. Therefore, all the remaining nodes at the first level, and hence the root node, can be fathomed as soon as the solution η^p to the 2-CKP (22)–(25) and (27) (where $R = N$) is such that

$$\sum_{j \in N} s_j \eta_j^p < \frac{\sum_{j \in N} s_j}{z_T}. \tag{29}$$

Clearly, the same dominance test can be applied at any level $l + 1$ of the tree, by replacing (29) by

$$\sum_{j \in R} s_j \eta_j^p < \frac{\sum_{j \in R} s_j}{z_T - l}. \tag{30}$$

This simple dominance test turns out to be very effective in some cases, allowing for a very small number of backtrackings to each node before its fathoming.

An exact algorithm very similar to BB_3 is a branch-and-price (see [1]) algorithm, called BP, based on the LP relaxation (7)–(9). At each node, a lower bound is computed by solving (7)–(9), amended by the branching constraints (see below). The tree exploration is of a depth-first type. Again the branching scheme tries to fill up the bins as much as possible, here with respect to the item weights given by the optimal dual solution π^*. This corresponds to setting up a bin with the items associated with a variable σ_S with minimum reduced cost. The value of this minimum is 0 at optimality, and is attained at least by all variables with nonzero value in the primal LP solution. In order to break ties, we give priority to the variables with higher LP value. The set of subproblems generated from a node, in the order in which they are explorated, corresponds then to the variables in formulation (7)–(10), ranked by low LP reduced costs and by high LP value when the reduced cost is 0. The dominance test described above (at the first level) has then the form

$$\sum_{j \in B_1} \pi_j^* \geqslant \frac{\sum_{j \in N} \pi_j^*}{z_T} = \frac{z^*}{z_T}$$

where z^* is the optimal solution value to (7)–(9), i.e., $L_B = \lceil z^* \rceil$. In other words, the subproblems generated by fixing to 1 a variable with reduced cost greater than $1 - z^*/z_T$ can be fathomed. Observe that, according to the branching rule and the depth-first exploration, the first "dive" of BP yields the heuristic solution provided by H_B, which is therefore a variant of BP where no backtracking is allowed.

The computational results presented in the next section clearly show that BP is the most robust exact algorithm for the problem; on the other hand the time required to solve the LP relaxation (7)–(9) is quite large in some cases. The best alternative to BP is BB_3, which is sometimes very fast, but which does not always terminate in a reasonable time. Therefore, we suggest an overall *Hybrid* approach for the solution of the problem, which applies, in turn, the following procedures: L_C, L_1, L_2, the six greedy heuristics, procedure 2REF to each of the six solutions produced, H_M followed by 2REF, BB_3 with time limit t_{BB_3}, L_B, H_B, and finally BP, stopping, of course, as soon as optimality is proven. The idea is to try to solve the problem quickly, and to resort to column generation if the attempt fails. As to the time limit t_{BB_3}, it can be tuned by observing the comparative performances of BB_3 and the algorithms based on column generation. In particular, since column generation takes longer as the number of items per bin increases, we propose to define t_{BB_3} as an increasing function of n and $n/\bar{z}$, where $\bar{z}$ is an estimate of the optimal solution value. In our implementation we set $t_{\mathrm{BB}_3} := n^2/2\bar{z}$, where $\bar{z} := \max\{L_C, L_1, L_2\}$.

6. Computational results

We have implemented the lower bounding procedures of Section 3, and the heuristic and exact algorithms of Sections 4 and 5, respectively, in FORTRAN 77.

The 2-CKP instances we encounter both for solving the LP relaxation (7)–(9), and for implementing the branching rule of algorithm BB_3, are solved by using Martello–Toth branch-and-bound algorithm TWOKP [23]. In the column generation phase, we initialize the incumbent solution value to 1, and interrupt the algorithm as soon as a solution of value greater than 1 (corresponding to a negative reduced cost variable) is found. For solving the 2-CKP further constrained by (26), we simply adapted procedure 2-CKP so as to fathom nodes where all the items in S^k have the associated variable fixed to 1.

We use algorithm SAP by Derigs [8] for the maximum-weight matching problems to be solved in algorithm H_M. This algorithm computes a minimum-weight perfect matching of a graph with an even number of nodes. To ensure the existence of a perfect matching in our graph, we add a dummy node if the number of nodes is odd, and connect this dummy node to all the others with edges having profit equal to 0.

The LP solver used for the LP relaxation (7)–(9) is CPLEX 3.0, which is a very fast and robust LP solver, capable of dealing with the degeneracy problems that sometimes arise in the solution of this relaxation.

Computing times are given in Digital DECStation 5000/240 CPU seconds — this machine has approximately the same speed as a PC 486/100.

6.1. Test bed instances

We have tested our codes on ten classes of randomly generated instances. In our implementation we considered integer values for the capacities and the weights. Recall that c and d are the capacity of the bins in the first and second dimension, respectively. The table below summarizes the characteristics of each class, which are described in detail below. The notation u.d.$[a,b]$ means *uniformly distributed* in the interval $[a,b]$.

Class	c	d	w_j $(j \in N)$	v_j $(j \in N)$
1	1000	1000	u.d.$[100,400]$	u.d.$[100,400]$
2	1000	1000	u.d.$[1,1000]$	u.d.$[1,1000]$
3	1000	1000	u.d.$[200,800]$	u.d.$[200,800]$
4	1000	1000	u.d.$[50,200]$	u.d.$[50,200]$
5	1000	1000	u.d.$[25,100]$	u.d.$[25,100]$
6	150	150	u.d.$[20,100]$	u.d.$[20,100]$
7	150	150	u.d.$[20,100]$	u.d.$[w_j-10, w_j+10]$
8	150	150	u.d.$[20,100]$	u.d.$[110-w_j, 130-w_j]$
9	See text	See text	u.d.$[100,400]$	u.d.$[100,400]$
10	100	100	See text	See text

In the first six classes, for each item $j \in N$, w_j and v_j are uniformly random values in $[\alpha,\beta]$. Classes 1–3 have been proposed by Spieksma in [25], and are such that each bin contains, on average, about 4, 2 and 2 items, respectively. In order to consider instances where more items per bin are packed, we have introduced two new classes: Classes 4 and 5, where each bin contains, on average, about 8 and 16 items, respectively. For BPP, the most difficult instances mentioned in the literature (see [22,10,26–29]) consider bin capacities equal to 150, and weights uniformly distributed in $[20,100]$. The analogous class for 2-DVPP is Class 6. For all the above classes, the two weights w_j and v_j associated with a given item j are generated according to independent distributions. In order to consider instances where a correlation exists, two new classes, Classes 7 and 8, have been derived from the most difficult of the previous classes, namely Class 6. Accordingly, in both Classes 7 and 8 we set $c=d=150$. In Class 7, for $j \in N$, we generate w_j uniformly random in $[20,100]$, and then v_j uniformly random in $[w_j-10, w_j+10]$, so the two item weights are correlated. In Class 8, on the other hand, for $j \in N$, we generate w_j uniformly random in $[20,100]$ and then v_j uniformly random in $[110-w_j, 130-w_j]$, so the two item weights are anti-correlated. Note that, by definition, at most two items in this latter class can be packed together into a bin, so the corresponding instances are easy to solve by matching. We considered these instances anyway to show that some approaches have difficulty in solving them to optimality. Instances in the last two classes are constructed in an artificial way to make them difficult. In Class 9, item weights are generated as in Class 1, value L is computed as $\max\{\lceil\sum_{j\in N} w_j/1000\rceil, \lceil\sum_{j\in N} v_j/1000\rceil\}$, and the bin capacities are defined as $c{:=}\lceil\sum_{j\in N} w_j/L\rceil$ and $d{:=}\lceil\sum_{j\in N} v_j/L\rceil$. In this way, either the optimal solution almost completely fills up all bins for at least one dimension, which is unlikely in practice, or lower bound L_C is not equal to the optimal value, a substantial difference with respect to Class 1 instances. Moreover, analogously to Class 1, lower bounds L_1 and L_2 (and hence L_H) are typically not tight. Instances of Class 10 are generated similarly to those proposed by Falkenauer [10] for BPP: bin capacities c and d are both set to 100, the number n of items is a multiple of 3, and for $k = 1,\ldots,n/3$, $w_{3k-2}, w_{3k-1}, v_{3k-2}, v_{3k-1}$ are uniformly random in $[25,50]$, while w_{3k} and v_{3k} are given the values $c - w_{3k-2} - w_{3k-1}$ and $d - v_{3k-2} - v_{3k-1}$, respectively. Therefore, the optimal solution value is given by $L_C = \sum_{j\in N} w_j/c = \sum_{j\in N} v_j/d$, corresponding to all bins completely filled-up for both dimensions.

For each class, we considered the values $n = 25, 50, 100, 200$ (for Class 10, $n = 24, 51, 99, 201$) and solved 10 instances for each n value.

6.2. Lower bounds

Table 1 reports a comparison of the various lower bounding procedures. In the table, Column *AvgSol* gives the average value of the optimal (or best found) solution value. Moreover, for each lower bound we report the average computing time (time), the average percentage error with respect to the optimum (%err), and the number of instances for which the lower bound value is equal to the optimum (#opt). For the instances that we could not solve to optimality, we set the optimum to the best lower bound in the evaluation of entries %err and #opt. In order to indicate whether the real optimum was found in all cases or not, we report in column #sol the number of instances that we could solve to optimality. Finally, we give the number of instances for which we were able to compute L_B within 100,000 CPU seconds (#end): if this number is less than 10, the values given in time, #opt, and %err refer to the instances for which L_B was computed. Bounds L_C, L_1 and L_2 are very fast to compute. In particular, the time required for L_C was less then one microsecond in almost all cases, while L_2 requires, on average, the same time as L_1 in almost all cases. The time required by L_H never exceeded 20 min for each of our instances. The time spent in the computation of L_B is typically quite large, and grows very fast with n: sometimes, we have not even been able to compute L_B, giving up after 100,000 CPU seconds. This is due to the difficulty of the LP subproblems and 2-CKP instances to be solved. As one might expect, the time performance of the column generation approach is worse for the instances where many items fit into a bin, see Classes 1, 4, 5 and 9. On the other hand, for each instance for which we know L_B and the corresponding optimal solution value (in particular, all the instances with $n \leqslant 100$), these values coincide.

Table 1
Comparison of various lower bounding procedures

Class	n	(# sol)	AvgSol	L_C			L_1			L_2			L_H			L_B		
				time	(# opt)	%err	time	(# opt)	%err	time	(# opt)	%err	time	(# opt)	%err	time (# end)	(# opt)	%err
1	25	10	7.1	0.000	(10)	0.00	0.000	(0)	85.89	0.000	(0)	49.29	0.020	(10)	0.00	5.500 (10)	(10)	0.00
1	50	10	13.3	0.000	(10)	0.00	0.000	(0)	92.46	0.005	(0)	61.13	0.960	(10)	0.00	73.655 (10)	(10)	0.00
1	100	10	26.0	0.000	(10)	0.00	0.002	(0)	96.15	0.008	(0)	66.50	16.498	(10)	0.00	755.263 (10)	(10)	0.00
1	200	10	51.3	0.000	(10)	0.00	0.002	(0)	98.05	0.030	(0)	73.12	273.659	(10)	0.00	5069.085 (10)	(10)	0.00
2	25	10	17.2	0.000	(0)	19.07	0.000	(5)	2.96	0.002	(10)	0.00	0.003	(10)	0.00	0.335 (10)	(10)	0.00
2	50	10	31.5	0.000	(0)	14.36	0.000	(5)	2.24	0.007	(9)	0.32	0.142	(9)	0.32	2.185 (10)	(10)	0.00
2	100	10	61.8	0.000	(0)	15.08	0.020	(6)	1.31	0.025	(9)	0.33	2.137	(9)	0.17	8.227 (10)	(10)	0.00
2	200	10	117.3	0.000	(0)	12.00	0.072	(5)	0.96	0.078	(9)	0.08	31.390	(10)	0.00	55.627 (10)	(10)	0.00
3	25	10	17.2	0.000	(0)	20.00	0.005	(5)	2.96	0.002	(10)	0.00	0.007	(10)	0.00	0.351 (10)	(10)	0.00
3	50	10	31.4	0.000	(0)	16.26	0.009	(5)	2.54	0.005	(10)	0.00	0.015	(10)	0.00	2.113 (10)	(10)	0.00
3	100	10	61.6	0.000	(0)	16.18	0.020	(6)	1.31	0.022	(10)	0.00	0.042	(10)	0.00	9.462 (10)	(10)	0.00
3	200	10	117.1	0.000	(0)	12.65	0.073	(5)	0.96	0.086	(10)	0.00	0.160	(10)	0.00	62.243 (10)	(10)	0.00
4	25	10	4.0	0.000	(10)	0.00	0.000	(0)	75.00	0.002	(0)	75.00	0.002	(10)	0.00	6.935 (10)	(10)	0.00
4	50	10	6.9	0.000	(10)	0.00	0.000	(0)	85.48	0.002	(0)	85.48	0.180	(10)	0.00	43.852 (10)	(10)	0.00
4	100	10	13.2	0.000	(10)	0.00	0.002	(0)	92.42	0.007	(0)	92.42	4.312	(10)	0.00	893.980 (10)	(10)	0.00
4	200	10	25.6	0.000	(10)	0.00	0.003	(0)	96.09	0.023	(0)	96.09	148.712	(10)	0.00	1950.257 (5)	(5)	0.00
5	25	10	2.0	0.000	(10)	0.00	0.000	(0)	50.00	0.000	(0)	50.00	0.000	(10)	0.00	6.505 (10)	(10)	0.00
5	50	10	3.9	0.000	(10)	0.00	0.000	(0)	74.17	0.005	(0)	74.17	0.007	(10)	0.00	19.407 (10)	(10)	0.00
5	100	10	7.0	0.000	(10)	0.00	0.000	(0)	85.71	0.010	(0)	85.71	0.012	(10)	0.00	143.186 (10)	(10)	0.00
5	200	9	13.0	0.000	(10)	0.00	0.000	(0)	92.31	0.023	(0)	92.31	73.383	(10)	0.00	1686.519 (8)	(8)	0.00

6	25	10	11.7	0.000	(3)	5.92	0.000	(3)	11.08	0.000	(5)	6.07	0.032	(7)	2.51	1.243 (10)	(10)	0.00
6	50	10	22.1	0.000	(2)	4.06	0.003	(1)	12.62	0.009	(1)	6.87	1.528	(3)	3.62	8.975 (10)	(10)	0.00
6	100	10	42.3	0.000	(2)	2.12	0.015	(0)	16.91	0.023	(0)	8.31	28.612	(2)	2.12	157.863 (10)	(10)	0.00
6	200	8	82.7	0.000	(2)	1.08	0.045	(0)	17.00	0.098	(0)	8.58	488.578	(2)	1.08	3229.202 (10)	(10)	0.00
7	25	10	10.9	0.000	(7)	2.51	0.000	(1)	25.99	0.000	(2)	10.15	0.038	(7)	2.51	1.540 (10)	(10)	0.00
7	50	10	21.1	0.000	(8)	0.85	0.000	(0)	17.66	0.003	(0)	12.92	0.760	(8)	0.85	9.882 (10)	(10)	0.00
7	100	10	40.9	0.000	(7)	0.71	0.012	(0)	20.56	0.010	(0)	14.63	16.762	(7)	0.71	133.877 (10)	(10)	0.00
7	200	10	81.1	0.000	(8)	0.24	0.036	(0)	18.72	0.045	(0)	16.50	305.193	(8)	0.24	1831.883 (10)	(10)	0.00
8	25	10	13.0	0.000	(0)	14.62	0.000	(0)	26.92	0.005	(10)	0.00	0.007	(10)	0.00	0.663 (10)	(10)	0.00
8	50	10	25.0	0.000	(0)	14.40	0.000	(0)	29.20	0.020	(10)	0.00	0.020	(10)	0.00	2.393 (10)	(10)	0.00
8	100	10	50.0	0.000	(0)	16.80	0.012	(0)	30.60	0.143	(10)	0.00	0.155	(10)	0.00	17.160 (10)	(10)	0.00
8	200	10	100.0	0.000	(0)	17.70	0.043	(0)	32.40	1.093	(10)	0.00	1.137	(10)	0.00	203.892 (10)	(10)	0.00
9	25	10	8.1	0.000	(0)	12.36	0.000	(0)	87.64	0.003	(0)	37.22	0.135	(0)	12.36	5.338 (10)	(10)	0.00
9	50	10	14.3	0.000	(0)	7.01	0.000	(0)	92.99	0.000	(0)	54.77	1.553	(0)	7.01	157.920 (10)	(10)	0.00
9	100	10	27.0	0.000	(0)	3.71	0.000	(0)	96.29	0.010	(0)	62.59	21.262	(0)	3.71	2550.768 (10)	(10)	0.00
9	200	0	51.3	0.000	(10)	0.00	0.002	(0)	98.05	0.028	(0)	70.05	303.557	(10)	0.00	65 078.516 (9)	(9)	0.00
10	24	10	8.0	0.000	(10)	0.00	0.000	(0)	87.50	0.003	(8)	2.50	0.127	(10)	0.00	2.003 (10)	(10)	0.00
10	51	10	17.0	0.000	(10)	0.00	0.000	(0)	94.12	0.005	(3)	7.65	2.542	(10)	0.00	15.565 (10)	(10)	0.00
10	99	10	33.0	0.000	(10)	0.00	0.002	(0)	96.97	0.019	(0)	10.30	39.973	(10)	0.00	287.412 (10)	(10)	0.00
10	201	0	67.0	0.002	(10)	0.00	0.002	(0)	98.51	0.100	(0)	12.84	831.112	(10)	0.00	1189.147 (10)	(10)	0.00

As well as in Class 10, lower bound L_C gives the optimal solution value for all the instances in Classes 1, 4 and 5, where on average more than 3 items fit into each bin in the optimal solution, and for a few instances in Class 7. Furthermore, L_C is always equal to L_B for the instances in Class 9 with $n = 200$, even if it is systematically worse for smaller values of n. L_2 yields consistently better values than L_1, and is equal to the optimum for all the instances in Classes 3 and 8, and for most instances in Class 2, i.e., for all instances where on average 2 items or less fit into a bin in the optimal solution. Lower bound L_H was too seldom better than $\max\{L_C, L_1, L_2\}$ to justify the amount of time required for its computation.

6.3. Heuristic algorithms

Various heuristic algorithms are compared in Table 2, where we give the same information as in Table 1. For the instances that we could not solve to optimality, we set the optimum to the value of the best solution found in the evaluation of entries %err and #opt. Again, we report in column #sol the number of instances that we could solve to optimality. We also give the number of instances for which algorithm H_B required less than 100,000 CPU seconds (#end). Columns 2FFD and 2FFD$_\mu$ give the results obtained by the corresponding heuristics, while column FFD – 2REF gives the overall time required and the best result obtained by these two heuristics, each followed by the application of procedure 2REF. Similarly, columns GREEDY and GREEDY – 2REF give the overall time required and the best solution obtained by applying the six greedy heuristics mentioned in Section 4, followed in the second case by procedure 2REF. Column H_M – 2REF gives the results of procedure H_M followed by procedure 2REF. We do not give the results obtained by applying procedure 2REF to the solutions provided by H_B as we never obtained an improvement of these solutions. All greedy algorithms are very fast and give solutions which are, on average, of equivalent quality, with the exception of Class 8, where 2FFD$_\mu$ is far better and always yields the optimal solution. The application of 2REF requires some time, but is justified by the fact that in some cases the final solution is optimal, or at least improved. Heuristic H_M requires a computing time which is comparable with that of GREEDY–2REF, and yields better solution values for the instances in Classes 6 and 7. As anticipated, H_B yields typically better solutions than other heuristics, especially for big n values, with the exception of Class 8, but requires a considerable time, typically in between t_{L_B} and $3t_{L_B}$, where t_{L_B} is the time spent in the computation of L_B, see Table 1.

6.4. Exact algorithms

Table 3 gives the results of the various exact algorithms we have tried, each with a time limit of 10,000 CPU seconds. For BP, our code does not check this time limit before the computation of L_B and H_B is completed, but gives up anyway after 100,000 CPU seconds. Column Spieksma gives the results of the exact algorithm described in

Table 2
Comparison of various heuristic algorithms

Class	n	(# sol)	AvgSol	2FFD			2FFD$_\mu$			FFD $-$ 2REF			GREEDY			GREEDY $-$ 2REF			H_M			H_M $-$ 2REF			H_B		
				time	(# opt)	%err	time	(# opt)	%err	time	(# opt)	%err	time	(# opt)	%err	time	(# opt)	%err	time	(# opt)	%err	time	(# opt)	%err	time (# end)	(# opt)	%err
1	25	10	7.1	0.000	(6)	5.71	0.002	(5)	7.14	0.092	(7)	4.29	0.003	(7)	4.29	0.317	(8)	2.86	0.065	(8)	2.86	0.097	(8)	2.86	7.668 (10)	(10)	0.00
1	50	10	13.3	0.003	(1)	6.82	0.002	(1)	6.82	0.158	(2)	6.11	0.032	(3)	5.39	0.792	(3)	5.39	0.602	(3)	5.39	0.707	(3)	5.39	116.260 (10)	(9)	0.77
1	100	10	26.0	0.010	(0)	6.94	0.002	(0)	8.13	0.403	(0)	5.40	0.080	(0)	6.94	2.598	(1)	5.40	8.788	(0)	5.40	9.116	(1)	5.02	1466.337 (10)	(9)	0.38
1	200	10	51.3	0.043	(0)	7.60	0.017	(0)	8.40	1.310	(0)	6.42	0.342	(0)	6.24	9.510	(0)	5.45	152.252	(0)	5.46	153.457	(0)	5.46	18 533.656 (10)	(9)	0.40
2	25	10	17.2	0.000	(10)	0.00	0.000	(10)	0.00	0.147	(10)	0.00	0.018	(10)	0.00	0.752	(10)	0.00	0.030	(10)	0.00	0.122	(10)	0.00	0.372 (10)	(10)	0.00
2	50	10	31.5	0.000	(9)	0.33	0.002	(9)	0.33	0.293	(9)	0.33	0.027	(9)	0.33	1.868	(10)	0.00	0.342	(10)	0.00	0.582	(10)	0.00	2.507 (10)	(9)	0.32
2	100	10	61.8	0.007	(10)	0.00	0.005	(10)	0.00	0.833	(10)	0.00	0.112	(10)	0.00	5.935	(10)	0.00	4.300	(10)	0.00	5.058	(10)	0.00	21.408 (10)	(9)	0.15
2	200	10	117.3	0.019	(8)	0.27	0.015	(10)	0.00	2.713	(10)	0.00	0.412	(10)	0.00	20.577	(10)	0.00	67.563	(10)	0.00	70.230	(10)	0.00	146.047 (10)	(10)	0.00
3	25	10	17.2	0.002	(10)	0.00	0.000	(10)	0.00	0.143	(10)	0.00	0.010	(10)	0.00	0.728	(10)	0.00	0.038	(10)	0.00	0.128	(10)	0.00	0.373 (10)	(10)	0.00
3	50	10	31.4	0.003	(10)	0.00	0.000	(10)	0.00	0.296	(10)	0.00	0.027	(10)	0.00	1.887	(10)	0.00	0.265	(10)	0.00	0.495	(10)	0.00	2.502 (10)	(9)	0.32
3	100	10	61.6	0.005	(10)	0.00	0.005	(10)	0.00	0.835	(10)	0.00	0.122	(10)	0.00	5.988	(10)	0.00	3.735	(10)	0.00	4.502	(10)	0.00	9.768 (10)	(9)	0.17
3	200	10	117.1	0.027	(6)	0.35	0.015	(10)	0.00	2.722	(10)	0.00	0.422	(10)	0.00	20.470	(10)	0.00	49.705	(10)	0.00	52.350	(10)	0.00	64.417 (10)	(9)	0.09
4	25	10	4.0	0.002	(10)	0.00	0.000	(10)	0.00	0.060	(10)	0.00	0.010	(10)	0.00	0.073	(10)	0.00	0.055	(10)	0.00	0.060	(10)	0.00	12.230 (10)	(10)	0.00
4	50	10	6.9	0.002	(7)	4.52	0.000	(7)	4.52	0.100	(8)	3.10	0.020	(8)	3.10	0.370	(8)	3.10	0.557	(8)	3.10	0.603	(8)	3.10	114.082 (10)	(10)	0.00
4	100	10	13.2	0.015	(3)	5.38	0.002	(2)	6.10	0.235	(4)	4.62	0.077	(4)	4.62	1.247	(6)	3.08	7.978	(4)	4.62	8.147	(6)	3.08	3197.110 (10)	(10)	0.00
4	200	10	25.6	0.028	(0)	5.45	0.010	(0)	6.23	0.652	(1)	3.91	0.278	(0)	4.28	4.600	(1)	3.51	127.327	(1)	3.51	128.060	(1)	3.51	7033.483 (3)	(3)	0.00
5	25	10	2.0	0.000	(10)	0.00	0.000	(10)	0.00	0.053	(10)	0.00	0.007	(10)	0.00	0.065	(10)	0.00	0.055	(10)	0.00	0.059	(10)	0.00	7.103 (10)	(10)	0.00
5	50	10	3.9	0.002	(9)	3.33	0.000	(9)	3.33	0.055	(9)	3.33	0.017	(10)	0.00	0.077	(10)	0.00	0.562	(9)	3.33	0.573	(9)	3.33	64.993 (10)	(10)	0.00
5	100	10	7.0	0.014	(10)	0.00	0.002	(9)	1.43	0.132	(10)	0.00	0.067	(10)	0.00	0.572	(10)	0.00	7.993	(10)	0.00	8.137	(10)	0.00	1343.978 (10)	(10)	0.00
5	200	9	13.0	0.028	(5)	3.85	0.005	(4)	4.62	0.360	(5)	3.85	0.262	(6)	3.08	2.297	(6)	3.08	128.303	(6)	3.08	128.987	(6)	3.08	40 212.059 (8)	(8)	0.00

Table 2. (*Continued*)

Class	n	(# sol)	AvgSol	2FFD			2FFD$_\mu$			FFD − 2REF			GREEDY			GREEDY − 2REF			H_M			H_M − 2REF			H_B		
				time	(# opt)	%err	time	(# opt)	%err	time	(# opt)	%err	time	(# opt)	%err	time	(# opt)	%err	time	(# opt)	%err	time	(# opt)	%err	time (# end)	(# opt)	%err
6	25	10	11.7	0.002	(7)	2.65	0.000	(7)	2.65	0.118	(9)	0.91	0.012	(9)	0.91	0.527	(10)	0.00	0.063	(9)	0.83	0.130	(9)	0.83	1.745 (10)	(9)	0.91
6	50	10	22.1	0.005	(4)	3.25	0.000	(3)	3.71	0.233	(6)	1.84	0.030	(4)	2.77	1.335	(7)	1.39	0.585	(7)	1.37	0.753	(7)	1.37	14.353 (10)	(9)	0.45
6	100	10	42.3	0.010	(0)	5.70	0.002	(0)	5.46	0.603	(0)	4.75	0.113	(0)	4.99	4.280	(0)	3.79	7.773	(6)	0.96	8.305	(6)	0.96	293.978 (10)	(9)	0.24
6	200	8	82.7	0.032	(0)	5.92	0.017	(0)	6.76	2.022	(0)	5.56	0.448	(0)	5.07	15.397	(0)	4.71	151.950	(1)	1.08	153.823	(2)	0.96	6653.845 (10)	(10)	0.00
7	25	10	10.9	0.005	(9)	1.00	0.000	(9)	1.00	0.107	(9)	1.00	0.012	(9)	1.00	0.435	(9)	1.00	0.073	(10)	0.00	0.125	(10)	0.00	2.272 (10)	(10)	0.00
7	50	10	21.1	0.007	(7)	1.53	0.000	(7)	1.53	0.213	(7)	1.53	0.027	(7)	1.53	1.178	(7)	1.53	0.612	(8)	1.03	0.760	(8)	1.03	16.343 (10)	(8)	0.95
7	100	10	40.9	0.012	(3)	1.95	0.005	(3)	1.70	0.545	(3)	1.70	0.112	(3)	1.70	3.802	(3)	1.70	10.142	(5)	1.21	10.627	(5)	1.21	279.373 (10)	(8)	0.49
7	200	10	81.1	0.035	(0)	1.85	0.012	(0)	1.85	1.921	(0)	1.85	0.363	(0)	1.73	14.063	(0)	1.73	278.505	(1)	1.24	280.308	(1)	1.24	3929.177 (10)	(9)	0.12
8	25	10	13.0	0.002	(4)	9.23	0.000	(10)	0.00	0.103	(10)	0.00	0.023	(10)	0.00	0.540	(10)	0.00	0.052	(10)	0.00	0.118	(10)	0.00	0.692 (10)	(9)	0.77
8	50	10	25.0	0.000	(0)	24.00	0.000	(10)	0.00	0.235	(10)	0.00	0.053	(10)	0.00	1.548	(10)	0.00	0.435	(10)	0.00	0.630	(10)	0.00	2.660 (10)	(7)	1.20
8	100	10	50.0	0.009	(1)	21.80	0.002	(10)	0.00	0.727	(10)	0.00	0.352	(10)	0.00	5.578	(10)	0.00	4.880	(10)	0.00	5.575	(10)	0.00	18.625 (10)	(7)	0.60
8	200	10	100.0	0.022	(1)	24.90	0.015	(10)	0.00	2.723	(10)	0.00	2.408	(10)	0.00	22.357	(10)	0.00	82.127	(10)	0.00	84.735	(10)	0.00	206.273 (10)	(7)	0.30
9	25	10	8.1	0.003	(9)	1.25	0.000	(8)	2.50	0.085	(9)	1.25	0.014	(10)	0.00	0.363	(10)	0.00	0.068	(10)	0.00	0.107	(10)	0.00	7.477 (10)	(10)	0.00
9	50	10	14.3	0.003	(1)	6.34	0.002	(2)	5.52	0.162	(5)	3.43	0.027	(4)	4.14	0.836	(8)	1.33	0.637	(9)	0.71	0.740	(9)	0.71	244.210 (10)	(10)	0.00
9	100	10	27.0	0.010	(0)	6.67	0.005	(0)	6.67	0.393	(0)	4.83	0.080	(0)	5.56	2.650	(0)	3.71	8.882	(0)	4.08	9.225	(0)	4.08	5567.198 (10)	(10)	0.00
9	200	0	51.3	0.010	(1)	3.79	0.012	(0)	3.77	1.287	(3)	2.66	0.268	(3)	2.84	9.565	(3)	2.09	164.920	(7)	1.37	166.138	(8)	1.18	37 894.184 (2)	(2)	0.00
10	24	10	8.0	0.002	(0)	15.00	0.000	(0)	21.25	0.090	(0)	13.75	0.009	(0)	15.00	0.367	(0)	12.50	0.016	(0)	15.00	0.403	(0)	12.50	2.045 (10)	(10)	0.00
10	51	10	17.0	0.002	(0)	12.35	0.003	(0)	13.53	0.185	(0)	11.76	0.032	(0)	11.76	1.052	(0)	9.41	0.063	(0)	11.76	1.157	(0)	9.41	16.020 (10)	(10)	0.00
10	99	10	33.0	0.010	(0)	9.09	0.010	(0)	10.30	0.493	(0)	9.09	0.127	(0)	9.09	3.365	(0)	7.27	5.023	(0)	3.03	5.423	(0)	3.03	290.432 (10)	(10)	0.00
10	201	0	67.0	0.037	(0)	6.32	0.014	(0)	6.91	1.710	(0)	6.18	0.486	(0)	6.03	12.658	(0)	5.59	63.215	(10)	0.00	64.700	(10)	0.00	2168.310 (10)	(10)	0.00

Table 3
Comparison of various exact algorithms

Class	n	Spieksma		BB_1				BB_2				BB_3				BP				Hybrid		
		time	(# opt)	time	(# opt)	# nodes	(# root)	time	(# opt)	# nodes	(# root)	time	(# opt)	# nodes	(# root)	time	(# opt)	# nodes	(# root)	time	(# opt)	# no *CG*
1	25	0.335	(10)	0.243	(10)	298	(8)	33.297	(10)	78 733	(8)	0.310	(10)	2	(8)	7.668	(10)	1	(10)	0.310	(10)	(10)
1	50	794.097	(7)	996.711	(6)	856 447	(3)	1051.221	(8)	1 052 401	(3)	30.452	(10)	124	(3)	129.503	(10)	2	(9)	30.452	(10)	(10)
1	100	—	(0)	1.883	(2)	24	(1)	10.809	(2)	93	(1)	109.727	(8)	20	(1)	1579.792	(10)	7	(9)	678.723	(10)	(7)
1	200	—	(—)	—	(0)	—	(0)	—	(0)	—	(0)	844.437	(5)	595	(0)	19 481.840	(9)	1	(9)	15 437.271	(10)	(4)
2	25	0.317	(10)	0.150	(10)	1	(10)	0.150	(10)	1	(10)	0.150	(10)	1	(10)	0.372	(10)	1	(10)	0.150	(10)	(10)
2	50	734.027	(7)	0.504	(9)	1	(9)	3.754	(10)	1903	(9)	1.577	(10)	2	(9)	2.538	(10)	5	(9)	1.577	(10)	(10)
2	100	0.185	(6)	0.888	(9)	1	(9)	0.888	(9)	1	(9)	0.888	(9)	1	(9)	21.408	(10)	8	(9)	3.037	(10)	(9)
2	200	—	(—)	9.195	(10)	1	(10)	9.195	(10)	1	(10)	9.195	(10)	1	(10)	146.047	(10)	1	(10)	9.195	(10)	(10)
3	25	0.224	(10)	0.152	(10)	1	(10)	0.152	(10)	1	(10)	0.152	(10)	1	(10)	0.373	(10)	1	(10)	0.152	(10)	(10)
3	50	1310.469	(9)	0.315	(10)	1	(10)	0.315	(10)	1	(10)	0.315	(10)	1	(10)	2.580	(10)	4	(9)	0.315	(10)	(10)
3	100	0.241	(6)	0.887	(10)	1	(10)	0.887	(10)	1	(10)	0.887	(10)	1	(10)	10.042	(10)	7	(9)	0.887	(10)	(10)
3	200	—	(—)	2.923	(10)	1	(10)	2.923	(10)	1	(10)	2.923	(10)	1	(10)	64.417	(10)	12	(9)	2.923	(10)	(10)
4	25	0.144	(10)	0.063	(10)	1	(10)	0.063	(10)	1	(10)	0.063	(10)	1	(10)	12.230	(10)	1	(10)	0.063	(10)	(10)
4	50	0.258	(10)	429.500	(9)	373 677	(8)	0.310	(10)	42	(8)	167.797	(10)	2	(8)	114.082	(10)	1	(10)	36.725	(10)	(9)
4	100	546.582	(5)	0.492	(6)	1	(6)	231.474	(9)	73 668	(6)	865.940	(8)	4	(6)	3197.110	(10)	1	(10)	1705.281	(10)	(7)
4	200	—	(—)	—	(0)	—	(0)	3614.175	(4)	257 765	(0)	178.004	(8)	19	(0)	7033.483	(3)	1	(3)	88.048	(7)	(7)
5	25	0.142	(10)	0.053	(10)	1	(10)	0.053	(10)	1	(10)	0.053	(10)	1	(10)	7.103	(10)	1	(10)	0.053	(10)	(10)
5	50	0.222	(10)	0.064	(10)	1	(10)	0.064	(10)	1	(10)	0.064	(10)	1	(10)	64.993	(10)	1	(10)	0.064	(10)	(10)
5	100	0.209	(10)	0.159	(10)	1	(10)	0.159	(10)	1	(10)	0.159	(10)	1	(10)	1343.978	(10)	1	(10)	0.159	(10)	(10)
5	200	—	(—)	0.750	(6)	10	(5)	57.453	(7)	870	(5)	69.771	(8)	4	(5)	40 212.059	(8)	1	(8)	7975.229	(9)	(8)

Table 3. (*Continued*)

Class	n	Spieksma		BB_1				BB_2				BB_3				BP				Hybrid		
		time	(# opt)	time	(# opt)	# nodes	(# root)	time	(# opt)	# nodes	(# root)	time	(# opt)	# nodes	(# root)	time	(# opt)	# nodes	(# root)	time	(# opt)	# no *CG*
6	25	20.364	(10)	5.595	(10)	11 284	(7)	39.244	(10)	79 591	(7)	28.819	(10)	1130	(7)	1.745	(10)	1	(9)	5.888	(10)	(8)
6	50	458.286	(2)	1502.921	(4)	672 974	(1)	978.800	(5)	493 439	(1)	1883.995	(3)	61 450	(1)	14.465	(10)	2	(9)	62.004	(10)	(1)
6	100	—	(0)	—	(0)	—	(0)	—	(0)	—	(0)	—	(0)	—	(0)	308.133	(10)	13	(9)	246.377	(10)	(0)
6	200	—	(—)	—	(—)	—	(—)	—	(—)	—	(—)	—	(—)	—	(—)	6565.469	(8)	1	(8)	6739.017	(8)	(0)
7	25	29.031	(10)	2.555	(10)	5417	(7)	0.883	(10)	938	(7)	13.608	(10)	1098	(7)	2.272	(10)	1	(10)	5.331	(10)	(9)
7	50	0.164	(5)	395.565	(8)	295 113	(6)	0.665	(7)	13	(6)	0.574	(7)	3	(6)	16.690	(10)	7	(8)	20.003	(10)	(7)
7	100	0.178	(2)	5.823	(3)	1	(3)	577.320	(6)	182 882	(3)	5.975	(4)	6	(3)	320.205	(10)	22	(8)	239.354	(10)	(4)
7	200	—	(—)	327.499	(1)	1	(1)	2773.021	(4)	198 915	(1)	188.816	(2)	41	(1)	4136.122	(10)	31	(9)	3844.824	(10)	(2)
8	25	1306.526	(2)	0.112	(10)	1	(10)	0.112	(10)	1	(10)	0.112	(10)	1	(10)	0.702	(10)	2	(9)	0.112	(10)	(10)
8	50	—	(0)	0.255	(10)	1	(10)	0.255	(10)	1	(10)	0.255	(10)	1	(10)	2.728	(10)	8	(7)	0.255	(10)	(10)
8	100	—	(—)	0.892	(10)	1	(10)	0.892	(10)	1	(10)	0.892	(10)	1	(10)	18.625	(10)	16	(7)	0.892	(10)	(10)
8	200	—	(—)	3.897	(10)	1	(10)	3.897	(10)	1	(10)	3.897	(10)	1	(10)	206.273	(10)	30	(7)	3.897	(10)	(10)
9	25	0.233	(10)	869.993	(10)	1 242 657	(0)	0.705	(10)	774	(0)	0.793	(10)	2	(0)	7.477	(10)	1	(10)	0.793	(10)	(10)
9	50	732.793	(10)	—	(0)	—	(0)	869.194	(6)	367 081	(0)	126.817	(9)	14	(0)	244.210	(10)	1	(10)	108.803	(10)	(7)
9	100	—	(0)	—	(—)	—	(—)	—	(0)	—	(0)	—	(0)	—	(0)	5567.198	(10)	1	(10)	5759.619	(10)	(0)
9	200	—	(—)	—	(—)	—	(—)	—	(—)	—	(—)	—	(—)	—	(—)	—	(0)	—	(0)	—	(0)	(0)
10	24	0.264	(10)	90.277	(10)	131 982	(0)	6.327	(10)	17 549	(0)	0.253	(10)	8	(0)	2.045	(10)	1	(10)	0.253	(10)	(10)
10	51	—	(0)	—	(0)	—	(0)	1115.717	(1)	918 773	(0)	176.623	(10)	1767	(0)	16.020	(10)	1	(10)	46.587	(10)	(6)
10	99	—	(—)	—	(—)	—	(—)	—	(0)	—	(0)	—	(0)	—	(0)	290.432	(10)	1	(10)	438.932	(10)	(0)
10	201	—	(—)	—	(—)	—	(—)	—	(—)	—	(—)	—	(—)	—	(—)	—	(0)	—	(0)	—	(0)	(0)

[25], for which we obtained the Pascal code from the author. The code was run on a PC Pentium 100, and the times converted into Digital DECStation 5000/240 CPU seconds by using a multiplicative factor of 2.3, according to the relative speed of the two machines, which we evaluated by running the same FORTRAN code on both of them. For each algorithm, we give the number of instances solved (#opt) and the average computing time (time). For all algorithms but Spieksma, we also report the average number of branching nodes (nodes), and the number of instances for which branching was not required (#root). All average values are computed with respect to the instances solved. Whenever the problem was solved to optimality by applying all the lower bounding procedures with the exception of L_B and all the heuristic algorithms with the exception of H_B, the number of branching nodes was 1 for BB$_1$, BB$_2$ and BB$_3$. Similarly, the number of nodes was considered 1 for BP whenever the problem was solved to optimality by L_B and H_B. If no instance of a certain class for a given value of n was solved by an exact algorithm, we did not apply the algorithm for the next n value of the same class. Also, we could not try Spieksma's algorithm for n greater than 100, due to the memory restrictions of the Pascal compiler. BB$_1$, BB$_2$ and BB$_3$ clearly outperform Spieksma. BB$_3$ is in most cases better than BB$_1$ and BB$_2$, but still cannot solve many of the big instances in Classes $1,6,7,9$ and 10. All the instances with n up to 100 are solved to optimality by BP, although BP is often much slower than BB$_3$ for the instances the latter can solve. The time required by BP is essentially the same as that required by H_B. Finally, in column Hybrid we report the results of the hybrid approach, showing that, on average, the results are significantly better than those of a pure column generation algorithm. We give the number of instances that were solved without using column generation (noCG). As a final remark, we observe that instances in Classes 2, 3 and 8 with much more than 200 items can be solved by procedure Hybrid, without using column generation. For example, instances with 1000 items are systematically solved in less than one minute.

Acknowledgements

This work was supported by MURST, Italy. We thank Mauro Caprara for his helpful discussions, and in particular for having inspired procedure CHECK_INCOMPATIBILITY in the Appendix. We are also grateful to Frits Spieksma who gave us his branch-and-bound code for 2-DVPP.

Appendix A.

We will now show how to solve, in $O(n \log n)$ time, the problem of checking whether a given solution of 2-DVPP is *minimal*, i.e., if it is not possible to merge the contents of two non-empty bins into a unique bin. Note that the problem can be stated as: given an item set S, test whether all item pairs $i,\ j \in S$ are *incompatible*, i.e., either $w_i + w_j > 1$ or $v_i + v_j > 1$. A clear observation is that given two items j,k such that $w_j \leqslant w_k$ and $v_j \leqslant v_k$, every item that is incompatible with j is incompatible also with k. In this case we say that k *dominates* j. The algorithm we propose keeps an ordered data structure D storing the weights of a set of items $C \subseteq S$ where no item is dominating. Data structure D is sorted according to decreasing w weights and increasing v weights. Initially, D contains only the first element of S; then the remaining items in S are considered in turn, incompatibility is checked with the previously considered items, and possible insertions/deletions are performed in D, according to the following scheme.

```
procedure CHECK INCOMPATIBILITY;
input:  an instance $(n,(w_j,v_j),j \in N)$ of 2-DVPP;
output: incompatible = TRUE or FALSE depending on whether all the item pairs
  are incompatible or not;
begin
 incompatible := FALSE;
 insert $(w_1,v_1)$ in $D$;
 for $j$:=2 to $n$ do
 begin
  find the item $s$ in $D$ such that $w_s$ is maximum and $w_s \leqslant 1 - w_j$;
  let $s$:=0 if no such item exists;
  if $s \neq 0$ and $v_s + v_j \leqslant 1$ then return;
  comment item $j$ is incompatible with items $1,\ldots,j-1$: update $D$;
  find the first item $s$ in $D$ such that $w_s \leqslant w_j$;
  find the last item $p$ in $D$ such that $w_p > w_j$;
  let $s$:=0 and $p$:=0 if the corresponding items do not exist;
  while $p \neq 0$ and $v_p \geqslant v_j$ do
  begin
   delete item $p$ from $D$;
   let $p$ be the predecessor of the item deleted in $D$;
   let $p$:=0 if no predecessor exists
  end
  if $s \neq 0$ then
  begin
   if $w_s = w_j$ and $v_s > v_j$ then
   begin
    delete item $s$ from $D$;
    insert item $j$ in $D$
   end
   else if $w_s < w_j$ and $v_s > v_j$ then
     insert item $j$ in $D$
  end
  else insert item $j$ in $D$
 end;
 incompatible := TRUE
end.
```

By implementing D using *red-black trees*, see, e.g., [7], it takes $O(\log k)$ time both for searching for the first element s with w_s not greater than a given threshold, and for inserting/deleting an element, where k is the number of elements stored. Also, finding the predecessor of an element takes constant time. Since the number of searches, insertions and deletions during the procedure is $O(n)$, the overall complexity is $O(n \log n)$.

References

[1] C. Barnhart, E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh, P.H. Vance, Branch-and-price: column generation for solving huge integer programs, Oper. Res. 46 (1998) 316–329.

[2] A. Caprara, Properties of some ILP formulations of a class of partitioning problems, Discrete Appl. Math. 87 (1998) 11–23.

[3] L.M.A. Chan, D. Simchi-Levi, J. Bramel, Worst-case analyses, linear programming and the bin packing problem, Math. Programming 83 (1998) 213–227.

[4] C. Chekuri, S. Khanna, On multi-dimensional packing problems, Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99), ACM Press, New York, 1999.

[5] B. Chen, B. Srivastava, An improved lower bound for the bin packing problem, Discrete Appl. Math. 66 (1996) 81–94.

[6] E.G. Coffman Jr., M.R. Garey, D.S. Johnson, Approximation algorithms for bin-packing — an updated survey, in: D.S. Hochbaum (Ed.), Approximation Algorithms for NP-Hard Problems, PWS Publishing Company, Boston, 1997.

[7] T.H. Cormen, C.E. Leiserson, R.R. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.

[8] U. Derigs, Solving non-bipartite matching problems via shortest path techniques, Ann. Oper. Res. 13 (1988) 225–261.

[9] S. Eilon, N. Christofides, The loading problem, Management Sci. 17 (1971) 259–267.

[10] E. Falkenauer, A hybrid grouping genetic algorithm for bin packing, J. Heuristics 2 (1996) 5–30.

[11] W. Fernandez de la Vega, G.S. Lueker, Bin-packing can be solved within $1 + \varepsilon$ in linear time, Combinatorica 1 (1981) 349–355.

[12] M.R. Garey, R.L. Graham, D.S. Johnson, A.C. Yao, Resource constrained scheduling as generalized bin packing, J. Combin. Theory (A) 21 (1976) 257–298.

[13] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, New York, 1979.

[14] A.M.H. Gerards, Matching, in: M.O. Ball, T.L. Magnanti, C.L. Monma, G.L. Nemhauser (Eds.), Handbook in Operations Research and Management Sciences, Vol.: Networks, North-Holland, Amsterdam, 1995.

[15] P.C. Gilmore, R.E. Gomory, A linear programming approach to the cutting stock problem, Oper. Res. 9 (1961) 849–859.

[16] P.C. Gilmore, R.E. Gomory, A linear programming approach to the cutting stock problem: Part II, Oper. Res. 11 (1963) 863–888.

[17] M. Grötschel, L. Lovász, A. Schrijver, The ellipsoid algorithm and its consequences in combinatorial optimization, Combinatorica 1 (1981) 169–197.

[18] P.L. Hammer, N.V.R. Mahadev, Bithreshold graphs, SIAM J. Algebraic Discrete Methods 6 (1985) 497–506.

[19] B.T. Han, G. Diehr, J.S. Cook, Multiple-type, two-dimensional bin packing problems: applications and algorithms, Ann. Oper. Res. 50 (1994) 239–261.

[20] M.S. Hung, J.R. Brown, An algorithm for a class of loading problems, Naval Res. Logistics Quart. 25 (1978) 289–297.

[21] S. Martello, P. Toth, Lower bounds and reduction procedures for the bin packing problem, Discrete Appl. Math. 28 (1990) 59–70.

[22] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, Wiley, New York, 1990.

[23] S. Martello, P. Toth, Upper bounds and algorithms for the two-constraint knapsack problem, Working paper, University of Bologna, 1996.

[24] K. Maruyama, S.K. Chang, D.T. Tang, A general packing algorithm for multidimensional resource requirements, Internat. J. Comput. Inform. Sci. 6 (1977) 131–149.

[25] F.C.R. Spieksma, A branch-and-bound algorithm for the two-dimensional vector packing problem, Comput. oper. Res. 21 (1994) 19–25.

[26] J.M. Valerio de Carvalho, Exact solution of cutting stock problems using column generation and branch-and-bound, International Transactions in Operational Research 5 (1998) 35–44.

[27] P.H. Vance, Branch-and-price algorithms for the one-dimensional cutting stock problem, Comput. Optim. Appl. 9 (1998) 211–228.

[28] P.H. Vance, C. Barnhart, E.L. Johnson, G.L. Nemhauser, Solving binary cutting stock problems by column generation and branch-and-bound, Comput. Optim. Appl. 3 (1994) 111–130.

[29] F. Vanderbeck, Computational study of a column generation algorithm for bin packing and cutting stock problems, Math. Programming 86 (1999) 565–594.

[30] G.J. Woeginger, There is no asymptotic PTAS for two-dimensional vector packing, Inform. Process. Lett. 64 (1997) 293–297.

[31] A.C. Yao, New algorithms for bin packing, J. ACM 27 (1980) 207–227.