Available online at www.sciencedirect.com

ScienceDirect

European Journal of Operational Research 191 (2008) 61–85

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

www.elsevier.com/locate/ejor

Discrete Optimization

Algorithms for two-dimensional cutting stock and strip packing problems using dynamic programming and column generation ☆

G.F. Cintra [a], F.K. Miyazawa [b], Y. Wakabayashi [c,*], E.C. Xavier [d]

[a] Centro Federal de Educação Tecnológica do Ceará, Av. 13 de Maio 2081, 60040-531, Fortaleza, CE, Brazil
[b] Instituto de Computação, Universidade Estadual de Campinas, Caixa Postal 6176, 13084-971, Campinas, SP, Brazil
[c] Instituto de Matemática e Estatística, Universidade de São Paulo, Rua do Matão 1010, 05508-090 São Paulo, SP, Brazil
[d] Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, São Paulo, SP, Brazil

Received 18 May 2006; accepted 14 August 2007
Available online 23 August 2007

Abstract

We investigate several two-dimensional guillotine cutting stock problems and their variants in which orthogonal rotations are allowed. We first present two dynamic programming based algorithms for the *Rectangular Knapsack* (RK) problem and its variants in which the patterns must be staged. The first algorithm solves the recurrence formula proposed by Beasley; the second algorithm – for staged patterns – also uses a recurrence formula. We show that if the items are not so small compared to the dimensions of the bin, then these algorithms require polynomial time. Using these algorithms we solved all instances of the RK problem found at the OR-LIBRARY, including one for which no optimal solution was known. We also consider the *Two-dimensional Cutting Stock* problem. We present a column generation based algorithm for this problem that uses the first algorithm above mentioned to generate the columns. We propose two strategies to tackle the residual instances. We also investigate a variant of this problem where the bins have different sizes. At last, we study the *Two-dimensional Strip Packing* problem. We also present a column generation based algorithm for this problem that uses the second algorithm above mentioned where staged patterns are imposed. In this case we solve instances for two-, three- and four-staged patterns. We report on some computational experiments with the various algorithms we propose in this paper. The results indicate that these algorithms seem to be suitable for solving real-world instances. We give a detailed description (a pseudo-code) of all the algorithms presented here, so that the reader may easily implement these algorithms.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Column generation; Cutting stock; Guillotine cutting; Dynamic programming; Two-dimensional packing; Strip packing

☆ This research was partially supported by ProNEx–FAPESP/CNPq (Proc. 2003/09925-5) and CNPq (Proc. 471460/04–4, 306526/04-2, 308138/04-0, 490333/04-4), CAPES.
* Corresponding author. Tel.: +55 11 30916052; fax: +55 11 3091 6134.
E-mail addresses: glauberc@cefetce.br (G.F. Cintra), fkm@ic.unicamp.br (F.K. Miyazawa), yw@ime.usp.br (Y. Wakabayashi), ecx@usp.br (E.C. Xavier).

0377-2217/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.ejor.2007.08.007

1. Introduction

Many industries face the challenge of finding solutions that are the most economical for the problem of cutting large objects to produce specified smaller objects. Very often, the large objects (bins) and the small objects (items) are two-dimensional and have rectangular shape. Besides that, a usual restriction for cutting problems is that in each object we may use only *guillotine cuts*, that is, cuts that are parallel to one of the sides of the object and go from one side to the opposite one; problems of this type are called two-dimensional guillotine cutting problems. Another usual restriction for these problems are the staged cuts. A *k-staged cutting* is a sequence of at most k stages of cuts, each stage of which is a set of parallel guillotine cuts performed on the objects obtained in the previous stage. Clearly, the cuts in each stage must be orthogonal to the cuts in the previous stage. We assume, without loss of generality, that the cuts are infinitely thin. In what follows, we define the problems we investigate in this paper. In all of them, even if it is not explicitly mentioned, only guillotine cuts are allowed.

In the *Rectangular Knapsack* (RK) problem we are given a rectangle $B = (W, H)$ of width W and height H, and a list of m items (types of rectangles), each item i of width w_i, height h_i, and value v_i $(i = 1,\dots,m)$. We wish to determine how to cut the rectangle B, so as to maximize the sum of the values of the items that are produced. We assume that many copies of the same item can be produced. We denote such an instance by $I = (W, H, w, h, v)$. Here, as well in the next problems, we assume that $w = (w_1,\dots,w_m)$, $h = (h_1,\dots,h_m)$, and $d = (d_1,\dots,d_m)$ are lists. According to the typology of Wäscher et al. [48], this problem corresponds to the `Two-dimensional Rectangular Single Large Object Packing Problem`.

The *Two-dimensional Cutting Stock* (2CS) problem is defined as follows. Given an unlimited quantity of two-dimensional bins $B = (W, H)$ of width W and height H, and a list of m items (small rectangles) each item i with dimensions (w_i, h_i) and demand d_i $(i = 1,\dots,m)$, determine how to cut the smallest number of bins B so as to produce d_i units of each item i. Such an instance for the 2CS problem is denoted by $I = (W, H, w, h, d)$. Following the typology of Wäscher et al. [48], this is the `Two-dimensional Rectangular Single Stock Size Cutting Stock Problem`.

We also consider the 2CS problem with variable bin sizes, denoted here as 2CSV. Wäscher et al. [48] refer to this problem as the `Two-dimensional Rectangular Multiple Stock Size Cutting Stock Problem`. The 2CSV problem is similar to the previous one: the difference is that we are now given a list of two-dimensional bin types $B_1,\dots,B_b$, each bin type B_j with dimensions (W_j, H_j) and value V_j (there is an unlimited quantity of them). We want to determine how to produce d_i units of each item i, $1 \leqslant i \leqslant m$, so as to minimize the sum of the values of the bins that are used. Such an instance for this problem is denoted by $I = (W, H, V, w, h, d)$, where $W = (W_1,\dots, W_b)$, $H = (H_1,\dots,H_b)$ and $V = (V_1,\dots,V_b)$.

The *Two-dimensional Strip Packing* (SP) problem is the following: given a two-dimensional strip of width W and infinite height, and a list of m items (rectangles), each item i with dimensions (w_i, h_i) and demand d_i, $1 \leqslant i \leqslant m$, determine how to produce d_i units of each item i from the strip, so as to minimize the height of the part of the strip that is used. We also require that the cuts be k-staged, and that in the first stage (in which horizontal cuts are performed) the distance between any two subsequent cuts be at most H (a restriction very common in practice, imposed by the cutting machines). An instance as above will be denoted by $I = (W, H, w, h, d)$. According to Wäscher et al. [48], this problem corresponds to the `Two-dimensional Rectangular Open Dimension Problem`.

For all these problems we consider variants with and without k-staged cuts and orthogonal rotations. Unless otherwise stated, we assume that the items are oriented (that is, rotations of the items are not allowed). The variants of these problems in which the items may be rotated orthogonally are denoted by RK^r, 2CS^r, 2CSV^r and SP^r. We also assume that, in all instances the items have feasible dimensions, that is, each of them fit into the given bin (or some bin type) or strip. We represent an empty list by $()$ and use the operator $\|$ to concatenate lists.

This paper focuses on algorithms for the problems above mentioned. They are classical hard optimization problems, interesting both from theoretical as well as practical point-of-view. Most of them have been largely investigated. In the following sections we discuss these problems and mention some of the results that have appeared in the literature.

We call each possible way of cutting a bin a *cutting pattern* (or simply *pattern*). To represent the patterns (and the cuts to be performed) we adopt the convention that is generally used in this context. We consider the Euclid-

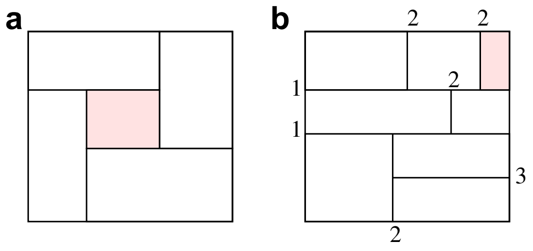

Fig. 1. (a) Non-guillotine pattern and (b) Guillotine pattern.

ean plane $\mathbb{R}^2$, with the *xy* coordinate system, and assume that the width of a rectangle is represented in the *x*-axis, and the height is represented in the *y*-axis. We also assume that the position (0,0) of this coordinate system represents the bottom left corner of the bin. Thus a bin of width W and height H corresponds to the region defined by the rectangle whose bottom left corner is at the position $(0,0)$ and the top right corner is at the position (W,H). To specify the position of an item i in the bin, we specify the coordinates of its bottom left corner.

A *guillotine pattern* is a pattern that can be obtained by a sequence of guillotine cuts applied to the original bin and to the subsequent small rectangles that are obtained after each cut (see Fig. 1). Many practical applications have restrictions on the number of cutting stages to obtain the final items, especially when the cost of the material to be cut is low compared to the industrial cost involved in the cutting process. We say that a pattern is *k-staged* if it is obtained after performing k stages of cutting (an eventual additional stage is allowed in order to separate an item from a wasted area). In Fig. 1b we have a 3-staged guillotine pattern (the gray area is a wasted area). Following other articles in the literature (see [12,13,46]) on problems on staged patterns, we assume that the first cutting stage is performed in the horizontal direction.

This paper is organized as follows. In Section 2, we focus on the Rectangular Knapsack (RK) problem, and also on two other variants of it: in the first, the items are allowed to be rotated orthogonally, and in the other, the patterns must be k-staged. We present dynamic programming based algorithms to obtain exact solutions for these problems. Section 3 is devoted to the Two-dimensional Cutting Stock (2CS) problem. We describe two algorithms for it, both based on the column generation approach. One of them uses a perturbation strategy we propose to deal with the residual instances. We also consider the variant of the 2CS problem in which orthogonal rotations are allowed. In Section 4 we study the 2CSV problem, a variant of the 2CS problem where bins may have different sizes and values. In Section 5 we study the Strip Packing (SP) problem. All algorithms based on the column generation approach we present here make use of the exact algorithms of Section 2.

For each one of these sections we report on the computational results we have obtained with the proposed algorithms. The tests indicate that for medium size instances the algorithms we describe here find in a short amount of time solutions that are very close to the optimum. All algorithms were implemented in C language. The computational tests were run on a computer with processor Intel Pentium IV, clock of 1.8 GHz, memory of 512 Mb and operating system *Linux* using the solver CLP (COIN-OR LP Solver) [22] as a linear system solver. For all problems we have performed tests considering guillotine patterns with and without orthogonal rotations and also with and without staged cuts. Finally, in Section 6 we make some final remarks concerning the performance of the algorithms and summarize our conclusions.

A preliminary version of part of this work appeared as an extended abstract in the proceedings of WEA 2004 [20]. We give here a more detailed description of the algorithms that appeared in [20] and also present algorithms for the 2CSV and SP problems, with and without staged patterns.

2. The Rectangular Knapsack problem

The *Rectangular Knapsack* (RK) problem has been largely investigated since the sixties. Gilmore and Gomory [27,28] studied this problem (on guillotine cuts) and also introduced in 1965 the variant on k-staged cuts [29]. In 1972, Herz [31] presented a recursive algorithm to obtain patterns, called canonical, making use of the so-called *discretization points*. Christofides and Whitlock [15] showed a dynamic programming approach to compute the discretization points. Some papers also consider exact tree search procedures [6,39] for this problem.

Beasley [5] proposed a dynamic programming approach using the discretization points of Herz for both the non-staged and the staged versions of the problem. Recently, Belov and Scheithauer [8] presented a branch-and-cut algorithm for a variant restricted to two-staged (oriented) patterns. Lodi and Monaci [36] also investigated the two-staged version. For the variant in which all items must be packed at most once, Jansen [33] obtained a $(2+\epsilon)$-approximation algorithm.

Before we describe the algorithms we implemented for the RK problem, we present first some concepts and results. We basically implemented the recurrence formulas proposed by Beasley combined with the concept of discretization points of Herz [31].

Let $I = (W, H, w, h, v)$ be an instance of the RK problem. We consider that W, H, and the entries of w and h are all integer numbers. If this is not the case, an equivalent integral instance can be obtained by an appropriate scaling.

A *discretization point of the width* (respectively, of the *height*) is a value $i \leqslant W$ (respectively, $j \leqslant H$) that can be obtained by an integer conic combination of $w_1, \ldots, w_m$ (respectively, $h_1, \ldots, h_m$). We denote by P (respectively, Q) the set of all discretization points of the width (respectively, height). Following Herz, we say that a *canonical pattern* is a pattern for which all cuts are made at discretization points.

We note that it suffices to consider only canonical patterns (for every pattern that is not canonical there is an equivalent one that is canonical). To refer to them, the following functions will be useful. For a rational $x \leqslant W$, let $p(x) := \max(i | i \in P, i \leqslant x)$ and for a rational $y \leqslant H$, let $q(y) := \max(j | j \in Q, j \leqslant y)$. Using these functions, it is not difficult to verify that the recurrence formula below, proposed by Beasley [5], can be used to calculate the value $V(w, h)$ of an optimal canonical guillotine pattern of a rectangle of dimensions (w, h). In this formula, $v(w, h)$ denotes the value of the most valuable item that can be cut in a rectangle of dimensions (w, h); it is 0 if no item can be cut in such a rectangle. Thus, $V(W, H)$ is the value of an optimal solution for an instance $I = (W, H, w, h, v)$.

$$V(w,h) = \max \left\{ \begin{array}{l} v(w,h) \\ \max\{V(w',h) + V(p(w-w'),h) | w' \in P \text{ and } 0 < w' \leqslant w/2\} \\ \max\{V(w,h') + V(w,q(h-h')) | h' \in Q \text{ and } 0 < h' \leqslant h/2\} \end{array} \right\}. \tag{1}$$

2.1. Discretization points

We present two algorithms to find the discretization points: the algorithms DEE (discretization by explicit enumeration) and DDP (discretization using dynamic programming). Both are described in the sequel.

In the algorithm DEE, D represents the width (or height) of the bin and $d_1, \ldots, d_m$ represent the widths (or heights) of the items. The algorithm DEE can be implemented to run in $O(m\delta)$ time, where δ represents the number of integer conic combinations of $d_1, \ldots, d_m$ with value at most D. This means that scaling does not affect the time required by DEE.

It is not difficult to construct instances for which an explicit enumeration may take exponential time. But if we can guarantee that $d_i > \frac{D}{k}$ $(i = 1, \ldots, m)$, then the sum of the m coefficients of any integer conic combination of $d_1, \ldots, d_m$ with value at most D is not greater than k. Thus, for a fixed k, the algorithm DEE is polynomial in m.

Algorithm 2.1 DEE

```
    Input: $D$ (width or height), $d_1, \ldots, d_m$.
    Output: a set $\mathscr{P}$ of discretization points (of the width or height).
$\mathscr{P} = \emptyset, k = 0$.
While $k \geqslant 0$ do
    For $i = k+1$ to $m$ do $z_i = \lfloor (D - \sum_{j=1}^{i-1} d_j z_j)/d_i \rfloor$.
    $\mathscr{P} = \mathscr{P} \cup \{\sum_{j=1}^{m} z_j d_j\}$.
    $k = \max(\{i | z_i > 0, 1 \leqslant i \leqslant m\} \cup \{-1\})$.
    If $k > 0$ then $z_k = z_k - 1$ and $\mathscr{P} = \mathscr{P} \cup \{\sum_{j=1}^{k} z_j d_j\}$.
Return $\mathscr{P}$.
```

In what follows we describe the algorithm DDP. The basic idea of this algorithm is to solve a knapsack problem in which every item i has weight and value d_i ($i = 1,\ldots,m$), and the knapsack has capacity D. The well-known dynamic programming technique for the knapsack problem (see [23]) finds optimal values of knapsacks with (integer) capacities taking values from 1 to D. It is easy to see that j is a discretization point if and only if the knapsack with capacity j has optimal value j.

Algorithm 2.2 DPP

```
  Input: $D, d_1, \ldots, d_m$.
  Output: a set $\mathscr{P}$ of discretization points.
$\mathscr{P} = \{0\}$.
**For** $j = 0$ **to** $D$ **do** $c_j = 0$.
**For** $i = 1$ **to** $m$ **do**
  **For** $j = d_i$ **to** $D$
    **If** $c_j < c_{j-d_i} + d_i$ **then** $c_j = c_{j-d_i} + d_i$.
**For** $j = 1$ **to** $D$
  **If** $c_j = j$ **then** $\mathscr{P} = \mathscr{P} \cup \{j\}$.
**Return** $\mathscr{P}$.
```

We note that the algorithm DDP requires time O(m D). Thus, the scaling (if needed) to obtain an integral instance may render the use of DDP unsuitable in practice. On the other hand, the algorithm DDP is suited for instances in which D is small. If D is large but the dimensions of the items are not so small compared to the dimensions of the bin, the algorithm DEE has a satisfactory performance. All the computational tests presented in Section 2.4 were performed with the algorithm DDP.

2.2. A dynamic programming algorithm for the RK problem

We describe now the algorithm DP (Algorithm 2.3) that solves the recurrence formula (1). Although there seems to be a straightforward way to solve the recurrence formula, we believe that the implementation we describe in what follows has shown to be very effective in practice.

Let $w_{\min}$ (respectively, $h_{\min}$) be the minimum width (respectively, height) of the items in the instance. Let P_0 be the set of values $i \in P$ such that $i \leqslant W - w_{\min}$, and let Q_0 be the set of values $j \in Q$ such that $j \leqslant H - h_{\min}$. Let $P_1 = P_0 \cup \{W\}$ and $Q_1 = Q_0 \cup \{H\}$. We can use the sets P_1 and Q_1 instead of the sets P and Q in the above recurrence and possibly obtain an improvement in the time to solve it, since no item can be to the right (respectively, to the top) of a vertical (respectively, horizontal) cut done in a position greater than $W - w_{\min}$ (respectively, $H - h_{\min}$).

We have designed this algorithm in such a way that a pattern corresponding to an optimal solution can be easily obtained. For that, the algorithm stores in a matrix, for every rectangle of width $p_i \in P_1$ and height $q_j \in Q_1$, which is the direction (horizontal or vertical) and the position of the first guillotine cut that has to be made in this rectangle. In case no cut should be made in the rectangle, the algorithm stores the item that corresponds to this rectangle.

The algorithm constructively solves the recurrence formula (1). The first step of the algorithm is to store for each rectangle of width $p_i \in P_1$ and height $q_j \in Q_1$, the item with largest value that can be packed on it (lines 1–5). At lines 6–17, the algorithm consider a rectangle of dimensions (p_i, q_j) and finds an optimal solution for this rectangle in the following manner: for each possible point p_x where a vertical cut can be done, the algorithm tests if the best known solution is worse than the one for which a vertical cut is done in the vertical direction at point p_x (lines 9–12). At lines 14–17 the algorithm tests a horizontal cut. The algorithm starts finding a solution with the smallest possible rectangle and then iteratively increases the size of the rectangle (lines 6–7), using the best known solutions of smaller rectangles to determine the best solution for the rectangle considered in the iteration.

Algorithm 2.3 DP

```
Input: An instance $I = (W, H, w, h, v)$ of the RK problem.
Output: An optimal solution for $I$.
Let $p_1 < \cdots < p_r$ be the points in the set $P_1$.
Let $q_1 < \cdots < q_s$ be the points in the set $Q_1$.
1 For $i = 1$ to $r$
2   For $j = 1$ to $s$
3     $V(i,j) = \max(\{v_k | 1 \leqslant k \leqslant m,\ w_k \leqslant p_i$ and $h_k \leqslant q_j\} \cup \{0\})$.
4     $item(i,j) = \max(\{k | 1 \leqslant k \leqslant m,\ w_k \leqslant p_i,\ h_k \leqslant q_j$ and $v_k = V(i,j)\} \cup \{0\})$.
5     $guillotine(i,j) = nil$.
6 For $i = 2$ to $r$
7   For $j = 2$ to $s$
8     $n = \max(k | 1 \leqslant k \leqslant i$ and $p_k \leqslant \lfloor \frac{p_i}{2} \rfloor)$.
9     For $x = 1$ to $n$
10      $t = \max(k | 1 \leqslant k \leqslant r$ and $p_k \leqslant p_i - p_x)$.
11      If $V(i,j) < V(x,j) + V(t,j)$ then
12        $V(i,j) = V(x,j) + V(t,j)$, $position(i,j) = p_x$ and $guillotine(i,j) =$ 'V'.
13    $n = \max(k | 1 \leqslant k \leqslant j$ and $q_k \leqslant \lfloor \frac{q_j}{2} \rfloor)$.
14    For $y = 1$ to $n$
15      $t = \max(k | 1 \leqslant k \leqslant s$ and $q_k \leqslant q_j - q_y)$.
16      If $V(i,j) < V(i,y) + V(i,t)$ then
17        $V(i,j) = V(i,y) + V(i,t)$, $position(i,j) = q_y$ and $guillotine(i,j) =$ 'H'.
```

When the algorithm DP halts, for each rectangle with dimensions (p_i, q_j), we have that $V(i,j)$ contains the optimal value that can be obtained for this rectangle; $guillotine(i,j)$ indicates the direction of the first guillotine cut, and $position(i,j)$ is the position (in the x-axis or in the y-axis) where the first guillotine cut has to be made. If $guillotine(i,j) = nil$, then no cut has to be made in this rectangle. In this case, $item(i,j)$ (if non-zero) indicates which item corresponds to this rectangle. The value of the optimal solution will be in $V(r, s)$.

Note that the assignments to variable t can be done in $O(\log r + \log s)$ time by performing a binary search in the set of the discretization points. If we use the algorithm DEE to calculate the discretization points, the algorithm DP can be implemented to have time complexity $O(m\delta_1 + m\delta_2 + r^2 s \log r + rs^2 \log s)$, where δ_1 and δ_2 represent the number of integer conic combinations that produce the discretization points of the width and of the height, respectively.

For instances with $w_i > \frac{W}{k}$ and $h_i > \frac{H}{k}$ (k fixed and $i = 1,\ldots,m$), we have that δ_1, δ_2, r and s are polynomial in m. Thus, for such instances the algorithm DP is polynomial in m.

We can use a vector X (resp. Y), of size W (resp. H), and let X_i (resp. Y_j) contain $p(i)$ (resp. $q(j)$). Once the discretization points are calculated, it requires time $O(W + H)$ to determine the values in the vectors X and Y. Using these vectors, each assignment to variable t can be done in constant time, and this leads to an implementation of the algorithm DP, using DEE (resp. DDP) as a subroutine, of time complexity $O(m\delta_1 + m\delta_2 + W + H + r^2 s + rs^2)$ (resp. $O(mW + mH + r^2 s + r s^2)$). In any case, the amount of memory required by the algorithm DP is $O(rs + W + H)$. We use this strategy in our implementation.

We can use the algorithm DP to solve the variant of the RK problem, denoted by RK^r, in which the items may be rotated orthogonally. For that, given an instance I of RK^r, we construct another instance (for RK) as follows. For each item i in I, of width w_i, height h_i and value v_i, we add another item of width h_i, height w_i and value v_i, whenever $w_i \neq h_i$, $w_i \leqslant H$ and $h_i \leqslant W$.

2.3. The k-staged RK problem

In this section we consider the RK and RK^r problems with k-staged cutting patterns. We present an exact algorithm for both problems.

Let $I = (W, H, w, h, v)$, with $w = (w_1,\ldots,w_m)$, $h = (h_1,\ldots, h_m)$ and $v = (v_1,\ldots,v_m)$, be an instance of the problem RK. We denote by $V(W,H,k,\mathscr{V})$, respectively, $V(W,H,k,\mathscr{H})$, the value of an optimal canonical guillotine k-staged pattern for a rectangle of dimensions (W, H). We use the following recurrence formulas to calculate these values. The parameters $\mathscr{H}$ and $\mathscr{V}$ indicate the direction of the first cutting stage: either horizontal or vertical:

$$V(w,h,0,\mathscr{V} \text{ or } \mathscr{H}) = v(w,h),$$
$$V(w,h,k,\mathscr{V}) = \max\{V(w,h,k-1,\mathscr{H}),(V(w',h,k-1,\mathscr{H})+V(p(w-w'),h,k,\mathscr{V})|w' \in P, w' \leqslant w/2\},$$
$$V(w,h,k,\mathscr{H}) = \max\{V(w,h,k-1,\mathscr{V}),(V(w,h',k-1,\mathscr{V})+V(w,q(h-h'),k,\mathscr{H})|h' \in Q, h' \leqslant h/2)\}.$$

As in Section 2, we use discretization points, and denote by P and Q the set of all discretization points of the width and height, respectively. We denote by $v(w, h)$ the value of the most valuable item that can be cut (or be obtained without any cut) from a rectangle of dimensions (w, h), or 0 if no item can be cut (or be obtained).

The algorithm SDP (Algorithm 2.4) that solves these recurrence formulas is described next. We also use in the algorithm, the sets P_1 and Q_1 defined in the last section.

The algorithm calculates the best solutions for the 1-staged problem and then uses this information to calculate the best solutions for the 2-staged problem and so forth (see the loop starting at line 7). For each one of these stages the algorithm works like the algorithm DP presented in the last subsection. The basic difference is that, for each stage, the algorithm only calculates solutions for cuts done in one direction; and two consecutive stages must have cuts in different directions. Since the first stage of cuts is done in the horizontal direction, when the number of stages is even (resp. odd), then the 1-staged subproblem must be solved with cuts in the vertical (resp. horizontal) direction (see line 6 of the algorithm). There may be a stage in which no cut has to be made: that happens when the best solution of a given stage, say l, is the best solution of the previous stage $l-1$. In this case, the value P is stored in the corresponding entry of *guillotine*, indicating that the solution is given by the previous stage (line 11).

Algorithm 2.4 SDP

```
Input: An instance $I = (W, H, w, h, v, k)$ of the $k$-staged RK problem.
Output: An optimal $k$-staged solution for $I$.
Let $p_1 < \cdots < p_r$, be the points in $P_1$.
Let $q_1 < \cdots < q_s$, be the points in $Q_1$.
1  For $i = 1$ to $r$
2    For $j = 1$ to $s$
3      $V(0, i, j) = \max(\{v_f | 1 \leqslant f \leqslant m, w_f \leqslant p_i \text{ and } h_f \leqslant q_j\} \cup \{0\})$.
4      $item(0, i, j) = \max(\{f | 1 \leqslant f \leqslant m, w_f \leqslant p_i, h_f \leqslant q_j \text{ and } v_f = V(0, i, j)\} \cup \{0\})$.
5      $guillotine(0,i,j) = nil$.
6  If $k$ is even then $A$ = 'H' else $A$ = 'V'
7  For $l = 1$ to $k$
8    For $i = 2$ to $r$
9      For $j = 2$ to $s$
10       $V(l, i, j) = V(l-1, i, j)$
11       $guillotine(l, i, j)$ = 'P'
12       If $A$ = 'V' then
13         $n = \max(f | 1 \leqslant f \leqslant s \text{ and } q_f \leqslant \lfloor \frac{q_j}{2} \rfloor)$.
14         For $y = 1$ to $n$
15           $t = \max(f | 1 \leqslant f \leqslant s \text{ and } q_f \leqslant q_j - q_y)$.
16           If $V(l, i, j) < V(l-1, i, y) + V(l, i, t)$ then
17           $V(l, i, j) = V(l-1, i, y) + V(l, i, t)$, $position(l, i, j) = q_y$ and $guillotine(l, i, j)$ = 'H'.
18       Else
19         $n = \max(f | 1 \leqslant f \leqslant r \text{ and } p_f \leqslant \lfloor \frac{p_i}{2} \rfloor)$.
20         For $x = 1$ to $n$
21           $t = \max(f | 1 \leqslant f \leqslant r \text{ and } p_f \leqslant p_i - p_x)$.
22           If $V(l, i, j) < V(l-1, x, j) + V(l, t, j)$ then
23             $V(l, i, j) = V(l-1, x, j) + V(l, t, j)$, $position(l, i, j) = p_x$ and $guillotine(l, i, j)$ = 'V'.
24   If $A$ = 'V' then $A$ = 'H' else $A$ = 'V'.
```

When the algorithm SDP halts, we have that $V(k, i, j)$ contains the optimal value that can be obtained in k stages for a rectangle with dimensions (p_i, q_j). Furthermore, *guillotine*(k, i, j) indicates the direction of the first guillotine cut, and *position*(k, i, j), stores the corresponding position of the first guillotine cut. In case no cut should be made in the rectangle, the algorithm stores the corresponding item in *item*(k, i, j).

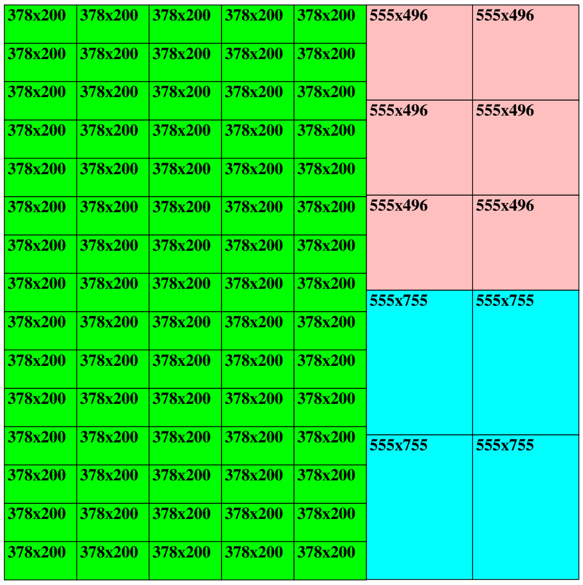

Fig. 2. The optimal solution for *gcut13* found by the algorithm DP. The small squares have dimensions (378, 200) and the other squares have dimensions (555, 496) and (555, 755).

It is not hard to see that the time complexity of this algorithm is the time complexity of the algorithm DP multiplied by k, the number of cutting stages. If we consider that k is limited by some constant then the overall time complexity remains the same.

We can also use the algorithm SDP to solve the k-staged RK^r problem. For that, we use the same technique of Section 2.2 where for each item i in I, of width w_i, height h_i and value v_i, we add another item of width h_i, height w_i and value v_i, whenever $w_i \neq h_i$, $w_i \leqslant H$ and $h_i \leqslant W$. We denote the corresponding algorithm for this case by SDP^r.

2.4. Computational results for the RK problem

The performance of the algorithm DP was tested with the instances of RK available in the OR-LIBRARY[1] (see [7] for a brief description of this library). We considered the 13 instances of RK, called *gcut1*,…,*gcut13* available in this library. For all these instances, with exception of instance *gcut13*, optimal solutions had already been found [5]. Using the algorithm DP we found an optimal solution for the instance *gcut13* in 22 seconds, shown in Fig. 2. In all these instances the value of each item is precisely its area. We note that Caprara and Monaci [14] and Fekete and Schepers [26] could not find an optimal solution for this instance in 1800 seconds in recent machines (a Pentium III 800 MHz and Pentium IV 2.8 GHz with 1 GB of memory, respectively). We recall that their approaches are for the more general setting in which the cuts need not be guillotine (in this general case, our approach can be used to obtain a lower bound).

Since the algorithm solved all these instances in a few seconds, we constructed other four instances (*gcut*14–*gcut*17) combining the instances available in the OR-LIBRARY. These new instances were obtained by putting together the items of the instance *gcut13* with the items of each of the instances *gcut9*, *gcut10*, *gcut11* and *gcul2*. For these new instances we considered that $B = (3500, 3500)$.

[1] http://mscmga.ms.ic.ac.uk/info.html

Table 1
Performance of the algorithm DP

Instance	Quantity of items	Dimensions of the bin	r	s	Optimal Solution	Waste (%)	Time (sec)
gcut1	10	(250, 250)	28	9	56,460	9.664	0
gcut2	20	(250, 250)	39	52	60,536	3.142	0
gcut3	30	(250, 250)	81	42	61,036	2.342	0
gcut4	50	(250, 250)	85	84	61,698	1.283	0
gcut5	10	(500, 500)	19	27	246,000	1.600	0
gcut6	20	(500, 500)	34	42	238,998	4.401	0
gcut7	30	(500, 500)	66	33	242,567	2.973	0
gcut8	50	(500, 500)	97	136	246,633	1.347	0
gcut9	10	(1000, 1000)	31	11	971,100	2.890	0
gcut10	20	(1000, 1000)	29	55	982,025	1.798	0
gcut11	30	(1000, 1000)	69	109	980,096	1.990	0
gcut12	50	(1000, 1000)	155	124	979,986	2.001	0
gcut13	32	(3000, 3000)	1457	2310	8,997,780	0.025	22.03
gcut14	42	(3500, 3500)	2390	2861	12,245,410	0.037	118.45
gcut15	52	(3500, 3500)	2422	2933	12,246,032	0.032	120.86
gcut16	62	(3500, 3500)	2559	2943	12,248,836	0.010	161.47
gcut17	82	(3500, 3500)	2676	2953	12,248,892	0.009	212.66

In Table 1 we show the instances solved and the computational results.

The column "Waste" shows – for each solution found – the percentage of the area of the bin that does not correspond to any item. The column "Time" indicates the time required to solve the instance; the entry 0 indicates that the time required is less than 0.000,001 seconds. Note that the new instances *gcut14*,...,*gcut17* turned out much harder to be solved: a few minutes were needed to find an optimal solution.

We also solved these instances with the algorithm SDP, for 2-, 3- and 4-staged cases. In 1985, Beasley [5] had already solved instances *gcut1*,...,*gcut12* for 2- and 3-staged patterns (but not the instance *gcut13*). In Table 2 we show the computational results only for 2- and 4-stages. We do not show the values of r and s, as they correspond to those shown in Table 1.

Table 2
Performance of the algorithm SDP for 2- and 4-staged patterns

Instance	Quantity of items	Dimensions of the bin	2-Staged			4-Staged		
			Optimal Solution	Waste (%)	(second)	Optimal Solution	Waste (%)	Time (sec)
gcut1	10	(250, 250)	56,460	9.66	0	56,460	9.66	0
gcut2	20	(250, 250)	60,076	3.878	0	60,536	3.142	0
gcut3	30	(250, 250)	60,133	3.787	0	61,036	2.342	0
gcut4	50	(250, 250)	61,698	1.283	0	61,698	1.283	0
gcut5	10	(500, 500)	246,000	1.600	0	246,000	1.600	0
gcut6	20	(500, 500)	235,058	5.977	0	238,998	4.401	0
gcut7	30	(500, 500)	242,567	2.973	0	242,567	2.973	0.017
gcut8	50	(500, 500)	245,758	1.697	0	246,633	1.347	0.071
gcut9	10	(1000, 1000)	971,100	2.890	0	971,100	2.890	0
gcut10	20	(1000, 1000)	982,025	1.798	0	982,025	1.798	0
gcut11	30	(1000, 1000)	974,638	2.536	0	980,096	1.990	0
gcut12	50	(1000, 1000)	977,768	2.223	0.01	979,986	2.001	0.01
gcut13	32	(3000, 3000)	8,906,216	1.042	21.82	8,997,780	0.025	43.72
gcut14	42	(3500, 3500)	12,216,788	0.271	124.55	12,242,100	0.064	264.41
gcut15	52	(3500, 3500)	12,215,614	0.281	137.21	12,242,100	0.064	289.98
gcut16	62	(3500, 3500)	12,210,837	0.320	177.21	12,244,511	0.045	371.60
gcut17	82	(3500, 3500)	12,232,948	0.139	223.13	12,246,694	0.027	456.00

Table 3
Performance of the algorithm DP with rotations

Instance	Quantity of items	Dimensions of the bin	r	s	Optimal Solution	Waste (%)	Time (sec)
gcut1r	10	(250, 250)	38	38	58,136	6.982	0
gcut2r	20	(250, 250)	77	77	60,611	3.022	0
gcut3r	30	(250, 250)	90	90	61,626	1.398	0
gcut4r	50	(250, 250)	105	105	62,265	0.376	0.01
gcut5r	10	(500, 500)	59	59	246,000	1.600	0
gcut6r	20	(500, 500)	79	79	240,951	3.620	0
gcut7r	30	(500, 500)	104	104	245,866	1.654	0
gcut8r	50	(500, 500)	166	166	247,787	0.885	0.02
gcut9r	10	(1000, 1000)	47	47	971,100	2.890	0
gcut10r	10	(1000, 1000)	94	94	982,025	1.798	0
gcut11r	30	(1000, 1000)	200	200	980,096	1.990	0.02
gcut12r	50	(1000, 1000)	258	258	988,694	1.131	0.03
gcut13r	32	(3000, 3000)	2354	2354	9,000,000	0.000	110.20
gcut14r	42	(3500, 3500)	2933	2933	12,250,000	0.000	306.35
gcut15r	52	(3500, 3500)	2953	2953	12,250,000	0.000	345.01
gcut16r	62	(3500, 3500)	2986	2986	12,250,000	0.000	354.91
gcut17r	82	(3500, 3500)	3007	3007	12,250,000	0.000	399.39

Table 4
Performance of the algorithm SDP for 2- and 4-staged patterns with rotations

Instances	Quantity of items	Dimensions of the bin	2-staged			4-staged		
			Optimal Solution	% Waste	Time (second)	Optimal Solution	% Waste	Time (sec)
gcut1r	10	(250, 250)	58,136	6.982	0	58,136	6.982	0
gcut2r	20	(250, 250)	60,611	3.022	0	60,611	3.022	0
gcut3r	30	(250, 250)	60,485	3.224	0	61,626	1.398	0
gcut4r	50	(250, 250)	62,265	0.376	0.01	62,265	0.376	0.01
gcut5r	10	(500, 500)	246,000	1.600	0	246,000	1.600	0
gcut6r	20	(500, 500)	240,951	3.620	0	240,951	3.620	0
gcut7r	30	(500, 500)	245,866	1.654	0.01	245,866	1.654	0
gcut8r	50	(500, 500)	247,260	1.096	0.01	247,787	0.885	0.02
gcut9r	10	(1000, 1000)	971,100	2.890	0	971,100	2.890	0
gcut10r	20	(1000, 1000)	982,025	1.798	0	982,025	1.798	0
gcut11r	30	(1000, 1000)	980,096	1.990	0.02	980,096	1.990	0.04
gcut12r	50	(1000, 1000)	988,694	1.131	0.03	988,694	1.131	0.06
gcut13r	32	(3000, 3000)	8,997,780	0.025	106.3	9,000,000	0.0	226.75
gcut14r	42	(3500, 3500)	12,240,515	0.077	322.77	12,247,796	0.018	702.19
gcut15r	52	(3500, 3500)	12,242,904	0.058	337.27	12,250,000	0.000	725.21
gcut16r	62	(3500, 3500)	12,243,100	0.056	368.20	12,250,000	0.000	800.55
gcut17r	82	(3500, 3500)	12,242,998	0.057	393.52	12,250,000	0.000	829.92

As one could expect, when we restrict to 2-staged patterns we have a slightly larger waste (compared to the results in Table 1). On the average, the waste for 2-staged patterns was less than 1% larger than the waste for non-staged patterns. Note that for the instances *gcut8*, *gcut13*...*gcut17* the wastes are very different. Moreover, for the instances *gcut1*...*gcut13* the solutions for 4-stages coincide with the optimal solutions for the unrestricted case.

We also run tests for the case in which rotations are allowed. For that, we considered the instances *gcut1*...*gcut17*, and named them as *gcut1r*,...,*gcut17r* (meaning that rotations are allowed). The performance of algorithms DP and SDP for these instances are presented in Tables 3 and 4. Comparing with the case without rotations, for some instances the time increased and the waste decreased (on the average less than

1%). For all these instances, except for the instance *gcut14r*, the solutions for the 4-staged case coincide with the optimal solutions for the unrestricted case.

3. The 2CS problem

We focus now on the *Two-dimensional Cutting Stock* (2CS) problem. Gilmore and Gomory [27–29] in the early sixties were the first to propose the use of the column generation approach for this problem. They proposed the k-staged pattern version and also considered the 2CSV problem, the variant of 2CS with bins of different sizes.

Alvarez-Valdes et al. [2] also presented a column generation approach for the 2CS problem. They used the dynamic programming algorithm presented by Beasley and also some meta-heuristic procedures. Puchinger and Raidl [42] investigated the 3-staged version: they applied the column generation approach using either a greedy heuristic or an evolutionary algorithm to generate columns.

Riehme et al. [44] designed an algorithm for the 2CS problem with *extremely varying order demands*. Their algorithm is also based on the column generation approach and is restricted to a 2-staged problem. Vanderbeck [46] also proposed a column generation approach for a cutting stock problem with several different restrictions: cuts must be 3-staged and unused parts of some stock can be used later as a new stock.

For the special case in which the demands are all equal to 1 (also known as bin packing problem), Chung et al. [16] presented the first approximation algorithm for this problem, called HFF (Hybrid First Fit), shown to have asymptotic performance bound at most 2.125. Later, Caprara [11] proved that HFF has asymptotic performance bound at most 2.077; and he also presented an 1.691-approximation algorithm (this is the best known result for this problem). These results are for the oriented case. When orthogonal rotations are allowed, Miyazawa and Wakabayashi [38] presented a 2.64-approximation algorithm. For the particular case in which all bins are squares and rotations are allowed, Epstein [25] presented a 2.45-approximation algorithm. In [19], we have shown that some of the approximation algorithms for the bin packing problem can be modified for the cutting stock problem. In this case the algorithms are of polynomial time and preserve the same approximation factor of the original algorithms. Recently, Puchinger and Raidl [43] presented a branch-and-price algorithm for the 3-staged two-dimensional bin packing problem. They also presented polynomial size formulations for this problem.

To discuss the column generation approach, let us first formulate the 2CS problem as an ILP (Integer Linear Program). Let $I = (W, H, w, h, d)$ be an instance for the 2CS problem. Represent each pattern j for the instance I as a vector p_j, whose ith entry indicates the number of times item i occurs in this pattern. The 2CS problem consists then in deciding how many times each pattern has to be used to meet the demands and minimize the total number of bins that are used.

Let n be the number of all possible patterns for I, and let P denote an $m \times n$ matrix whose columns are the patterns $p_1, \ldots, p_n$. If we denote by d the vector of the demands, then we have the following ILP formulation: minimize $\sum_{j=1}^{n} x_j$ subject to $Px = d$ and $x_j \geqslant 0$ and x_j integer for $j = 1, \ldots, n$. (The variable x_j indicates how many times the pattern j is selected.)

The well-known column generation method proposed by Gilmore and Gomory [27] consists in solving the relaxation of the above ILP, shown below. The idea is to start with a small set of columns of P and then generate new ones only when they are needed.

$$\begin{aligned} \text{minimize} \quad & x_1 + \cdots + x_n \\ \text{subject to} \quad & Px = d \\ & x_j \geqslant 0 \quad j = 1, \ldots, n. \end{aligned} \tag{2}$$

We can use an algorithm $\mathscr{R}$ (for the RK problem) to generate new columns (guillotine patterns). Note that, if each item i has value y_i and occurs z_i times in a pattern produced by $\mathscr{R}$, then $\sum_{i=1}^{m} y_i z_i$ is maximum. This is exactly what we need to generate new columns. We describe in the sequel the algorithm SimplexCG$_2$ that solves (2) (Algorithm 3.1).

Algorithm 3.1 SimplexCG$_2$

```
  *Input*: An instance $I = (W, H, w, h, d)$ of the 2CS problem.
  *Output*: An optimal solution for (2), where the columns of $P$ are patterns for $I$.
  *Subroutine*: An algorithm $\mathcal{R}$ for the RK problem.
1 Let $x = d$ and $B$ be the identity matrix of order $m$.
2 Solve $y^{\mathrm{T}}B = \mathbb{1}^{\mathrm{T}}$.
3 Generate a new column $z$ executing the algorithm $\mathcal{R}$ with parameters $W, H, w, h, y$.
4 **If** $y^{\mathrm{T}}z \leqslant 1$ **then** return $B$ and $x$ and halt ($x$ corresponds to the columns of $B$).
5 **Else** solve $Bw = z$.
6 Let $t = \min\left(\frac{x_j}{w_j} | 1 \leqslant j \leqslant m, w_j > 0\right)$.
7 Let $s = \min\left(j | 1 \leqslant j \leqslant m, \frac{x_j}{w_j} = t\right)$.
8 **For** $i = 1$ **to** $m$ **do**
  8.1 $B_{i,s} = z_i$.
  8.2 **If** $i = s$ **then** $x_i = t$ **else** $x_i = x_i - w_i t$.
9 **Go to** step 2.
```

We implemented this algorithm and for the subroutine $\mathcal{R}$ we used either the algorithm DP or the algorithm SDP (described in Section 2). We remark that in steps 2 and 5 we use the COIN-CLP [22] solver as the linear system solver. The computational tests indicated that on the average the number of columns generated by SimplexCG$_2$ was $O(m^2)$. This is in accordance with the theoretical results that are known with respect to the average behavior of the Simplex method [1,9]. Now using the (possibly fractional) solution obtained by SimplexCG$_2$, we can find an integer solution. For that, we developed the algorithm CG (see Algorithm 3.1) described in what follows.

Algorithm 3.2 CG

```
  *Input*: An instance $I = (W, H, w, h, d)$ of the 2CS problem.
  *Output*: A solution for $I$.
1 Execute the algorithm SimplexCG$_2$ with parameters $W, H, w, h, d$ obtaining $B$ and $x$.
2 **For** $i = 1$ **to** $m$ **do** $x^*_i = \lfloor x_i \rfloor$.
3 **If** $x^*_i > 0$ for some $i$, $1 \leqslant i \leqslant m$, **then**
  3.1 **Return** $B$ and $x^*_1, \ldots, x^*_m$ (but do not halt).
  3.2 **For** $i = 1$ **to** $m$ **do**
    3.2.1 **For** $j = 1$ **to** $m$ **do** $d_i = d_i - B_{i,j}x^*_j$.
  3.3 Let $m' = 0$, $w' = ()$, $h' = ()$ and $d' = ()$.
  3.4 **For** $i = 1$ **to** $m$ **do**
    3.4.1 **If** $d_i > 0$ **then** $m' = m' + 1$, $w' = w' \| (w_i)$, $h' = h' \| (h_i)$ and $d' = d' \| (d_i)$.
  3.5 **If** $m' = 0$ **then** halt.
  3.6 Let $m = m'$, $w = w'$, $h = h'$, $d = d'$ and **go to** step 1.
4 **Return** the solution of algorithm M-HFF executed with parameters $W, H, w, h, d$.
```

The algorithm is iterative. Each iteration starts with an instance I of the 2CS problem and consists basically in solving (2) with SimplexCG$_2$ obtaining B and x (step 1). In step 2 the algorithm calculates $x^* = (x^*_1, \ldots, x^*_m)$, where $x^*_i = \lfloor x_i \rfloor$ $(i = 1, \ldots, m)$. The algorithm returns B and x^* (step 3.1) as the current solution (possibly infeasible). The columns of B represent patterns, and x^* indicates how many times each one of these patterns is used in this solution. The demand of each item i that is not fulfilled is $d^*_i = d_i - \sum_{j=1}^{m} B_{i,j}x^*_j$. Thus, if we take $d^* = (d^*_1, \ldots, d^*_m)$, we have a residual instance $I^* = (W, H, w, h, d^*)$ (we may eliminate from I^* the items with no demand). The residual instance is calculated in steps 3.3 and 3.4. If all demands are fulfilled by the solution x^*, then the algorithm halts (step 3.5). Otherwise the algorithm solves again the linear program corresponding to the residual instance (step 3.6).

Notice that, in some iteration it might happen that the solution found by the algorithm SimplexCG$_2$ has all x_i values smaller than 1. In this case, $x^*_i = 0$ for all $i \in \{1, \ldots, m\}$, and thus (see step 3), we solve the instance I with the algorithm M-HFF (Modified HFF) that corresponds to the algorithm HFF modified to consider demands for the items, see [19]. We observe that the algorithm M-HFF can be implemented to run in polynomial time. As its asymptotic performance bound is at most 2.077 (see [11]), we may expect that using M-HFF we produce solutions of good quality.

Since in each iteration either part of the demand is fulfilled or we go to step 4, it follows that after a finite number of iterations the demands will be satisfied. In fact, it is easy to prove that step 3.6 of the algorithm CG is executed at most m times.

We note that the algorithm CG can be used to solve the variant of 2CS, called 2CS^r, in which orthogonal rotations of the items are allowed. For that, before we call the algorithm $\mathscr{R}$, in step 3 of SimplexCG$_2$, it suffices to make the transformation explained at the end of Section 2.2. We will call SimplexCG$_2^r$ the variant of SimplexCG$_2$ with this transformation. It should be noted however that the algorithm M-HFF, called in step 6 of CG, does not use the fact that the items can be rotated.

We designed a simple algorithm for the variant of 2CS^r in which all items have demand 1. This algorithm, called *First Fit Decreasing Height using Rotations* (FFDHR), has asymptotic approximation bound at most 4, as we have shown in [18]. Substituting the call to M-HFF with a call to FFDHR, we obtain the algorithm CGR, that is a specialized version of CG for the 2CS^r problem.

We also tested another modification of the algorithm CG (and of CGR). This is the following: when we solve an instance, and the solution returned by SimplexCG$_2$ rounded down is equal to zero, instead of simply submitting this instance to M-HFF (or FFDHR), we use M-HFF (or FFDHR) to obtain a *good* pattern, and update the demands; if there is some item for which the demand is not fulfilled, we go to step 1.

Note that, the basic idea is to *perturb* the residual instances whose relaxed LP solution, rounded down, is equal to zero. With this procedure, it is expected that the solution obtained by SimplexCG$_2$ for the residual instance has more variables with value greater than 1. The algorithm CG^p, described in what follows (Algorithm 3.3), incorporates this modification.

Algorithm 3.3 CG^p

```
  Input: An instance $I = (W, H, w, h, d)$ of 2CS$^r$.
  Output: A solution for $I$.
1 Execute the algorithm SimplexCG$_2$ with parameters $W, H, w, h, d$ obtaining $B$ and $x$.
2 For $i = 1$ to $m$ do $x_i^* = \lfloor x_i \rfloor$.
3 If $x_i^* > 0$ for some $i$, $1 \leqslant i \leqslant m$, then
    3.1 Return $B$ and $x_1^* \dots , x_m^*$ (but do not halt).
    3.2 For $i = 1$ to $m$ do
        3.2.1 For $j = 1$ to $m$ do $d_i = d_i - B_{i,j}x_j^*$.
    3.3 Let $m' = 0$, $w' = ()$, $h' = ()$ and $d' = ()$.
    3.4 For $i = 1$ to $m$ do
        3.4.1 If $d_i > 0$ then $m' = m' + 1$, $w' = w'\|(w_i)$, $h' = h'\|(h_i)$ and $d' = d'\|(d_i)$.
    3.5 If $m' = 0$ then halt.
    3.6 Let $m = m'$, $w = w'$, $h = h'$, $d = d'$ and go to step 1.
4 Return a pattern generated by the algorithm M-HFF, executed with parameters $W, H, w, h, d$, that has the smallest wasted area,
  and update the demands.
5 If there are demands to be fulfilled, go to step 1.
```

It should be noted that with this modification we cannot guarantee anymore that we have to make at most $m + 1$ calls to SimplexCG$_2$. It is however, easy to see that the algorithm CG^p in fact halts, as each time step 1 is executed the demand decreases strictly. After a finite number of iterations the demand will be fulfilled and the algorithm halts.

3.1. Computational results for the 2CS problem

We did not find instances for the 2CS problem in the OR-LIBRARY. We tested the algorithms CG and CG^p with the instances *gcut1*,…,*gcut12*, associating with each item i a randomly generated demand d_i between 1 and 100 (varying demands). We called these instances *gcut1d*,…,*gcut12d*.

We show in Table 5 the computational results obtained with the algorithm CG executed with subroutine DP. In this table, LB denotes the lower bound (given by the rounded up solution of (2)) for the value of an optimal integer solution.

Table 5
Performance of the algorithm CG

Instance	Solution of CG	LB	Difference from LB (%)	Time (sec)	Columns Generated	Solution of M-HFF	Improvement over M-HFF (%)
gcut1d	294	294.0	0.000	0.03	24	322	8.70
gcut2d	345	345.0	0.000	0.28	91	360	4.17
gcut3d	333	332.0	0.301	0.77	314	374	10.96
gcut4d	837	836.0	0.120	5.09	1031	878	4.67
gcut5d	198	197.0	0.508	0.03	29	224	11.61
gcut6d	344	343.0	0.292	0.21	115	395	12.91
gcut7d	592	591.0	0.169	0.33	158	642	7.79
gcut8d	692	690.0	0.290	3.60	670	765	9.54
gcut9d	132	131.0	0.763	0.06	44	141	6.38
gcut10d	293	293.0	0.000	0.09	47	328	10.67
gcut11d	331	330.0	0.303	1.07	358	375	11.73
gcut12d	672	672.0	0.000	3.35	674	722	6.93

The algorithm CG found optimal or quasi-optimal solutions for all these instances. On the average, the difference between the solution found by CG and the lower bound (LB) was only 0.228%. We note also that the time spent to solve these instances was satisfactory. Moreover, the gain of the solution found by CG compared to the solution found by M-HFF was 8.83%, on the average, a very significant improvement.

We have also used the algorithm CG^p to solve the instances *gcut1d*,...,*gcut12d*. The table with these results will be omitted, as the quality of the solutions that we obtained was exactly the same (we noted only a slight increase of time for some instances). But for the case with rotations, the algorithm CG^p gave better results.

The tests on instances where rotations are allowed (called *gcut1dr*...,*gcut12dr*) are presented in Table 6, for algorithm CGR; and in Table 7, for its version where a perturbation strategy is used (algorithm CGR^p).

The algorithm CGR found optimal or quasi-optimal solutions for all instances. The difference between the value found by CGR and the lower bound (LB) was only 0.31%, on the average. Comparing the value of the solutions obtained by CGR with the solutions obtained by FFDHR, we note that there was an improvement of 10.40%, on the average. This improvement would be of 15.11% if compared with the solution obtained by M-HFF.

Comparing CGR with CGR^p, we see that the algorithm CGR^p obtained an optimal solution for the instance *gcut12dr* and that the time it spent is very close to the time spent by CGR. The algorithm CGR^p also obtained better solutions for the instances *gcut3dr* and *gcut7dr*.

For the k-staged version, we show in Tables 8 and 9 the results obtained by the algorithm CG^p with subroutine SDP and $k = 2, 4$ (we omitted the solutions for $k = 3$, as they were very similar to those for $k = 4$). As one can see, the algorithm CG^p found solutions in a very short amount of time. We omit here the results obtained by

Table 6
Performance of the algorithm CGR

Instance	Solution of CG	LB	Difference from LB (%)	Time (sec)	Columns generated	Solution of FFDHR	Improvement over FFDHR (%)
gcut1dr	291	291.0	0.000	0.04	25	291	0.00
gcut2dr	283	282.0	0.355	1.41	174	314	9.87
gcut3dr	315	313.0	0.639	1.89	450	347	9.22
gcut4dr	836	836.0	0.000	4.40	662	846	1.18
gcut5dr	175	174.0	0.575	0.06	34	198	11.62
gcut6dr	302	301.0	0.332	0.25	94	371	18.60
gcut7dr	544	542.0	0.369	0.99	257	623	12.68
gcut8dr	651	650.0	0.154	6.47	625	734	11.31
gcut9dr	123	122.0	0.820	0.07	39	143	13.99
gcut10dr	270	270.0	0.000	0.21	68	301	10.30
gcut11dr	299	298.0	0.336	6.00	512	342	12.57
gcut12dr	602	601.0	0.166	19.24	803	696	13.51

Table 7
Performance of the algorithm CGR^p

Instance	Solution of CGR^p	LB	Difference from LB (%)	Time (sec)	Columns generated	Solution of FFDHR	Improvement over FFDHR (%)
gcut1dr	291	291.0	0.000	0.06	30	291	0.00
gcut2dr	283	282.0	0.355	1.57	214	314	9.87
gcut3dr	314	313.0	0.319	3.38	918	347	9.51
gcut4dr	836	836.0	0.000	6.82	1172	846	1.18
gcut5dr	175	174.0	0.575	0.10	61	198	11.62
gcut6dr	302	301.0	0.332	0.28	117	371	18.60
gcut7dr	543	542.0	0.185	1.18	357	623	12.84
gcut8dr	651	650.0	0.154	7.10	811	734	11.31
gcut9dr	123	122.0	0.820	0.10	53	143	13.99
gcut10dr	270	270.0	0.000	0.26	93	301	10.30
gcut11dr	299	298.0	0.336	6.47	674	342	12.57
gcut12dr	601	601.0	0.000	19.57	965	696	13.65

Table 8
Performance of the algorithm CG^p with 2-staged patterns

Instance	Solution of CG^p	LB	Difference from LB (%)	Time (sec)	Columns generated	Solution of M-HFF	Improvement over M-HFF (%)
gcut1d	295	295.0	0.000	0.03	21	322	8.39
gcut2d	345	345.0	0.000	0.45	173	360	4.17
gcut3d	343	342.0	0.292	1.31	534	374	8.29
gcut4d	845	845.0	0.000	5.99	1506	878	3.76
gcut5d	207	207.0	0.000	0.03	21	224	7.59
gcut6d	375	375.0	0.000	0.16	86	395	5.06
gcut7d	600	600.0	0.000	0.71	357	642	6.54
gcut8d	720	720.0	0.000	3.50	693	765	5.88
gcut9d	135	135.0	0.000	0.08	57	141	4.26
gcut10d	315	315.0	0.000	0.21	122	328	3.96
gcut11d	349	349.0	0.000	0.79	289	375	6.93
gcut12d	676	675.0	0.148	5.28	1167	722	6.37

the algorithm CG with subroutine SDP, since the algorithm CG^p found better results in most of the instances. We also present only the results of the algorithms with the perturbed method since they had better performance.

Table 9
Performance of the algorithm CG^p with 4-staged patterns

Instance	Solution of CG^p	LB	Difference from LB (%)	Time (sec)	Columns Generated	Solution of M-HFF	Improvement over M-HFF (%)
gcut1d	294	294.0	0.000	0.04	25	322	8.70
gcut2d	345	345.0	0.000	0.49	157	360	4.17
gcut3d	333	332.0	0.301	1.76	621	374	10.96
gcut4d	837	836.0	0.120	7.27	1606	878	4.67
gcut5d	198	197.0	0.508	0.06	40	224	11.61
gcut6d	344	343.0	0.292	0.26	136	395	12.91
gcut7d	592	591.0	0.169	0.61	295	642	7.79
gcut8d	691	690.0	0.145	10.33	1539	765	9.67
gcut9d	131	131.0	0.000	0.08	55	141	7.09
gcut10d	294	293.0	0.341	0.17	93	328	10.37
gcut11d	330	330.0	0.000	1.88	535	375	12.00
gcut12d	673	672.0	0.149	5.70	927	722	6.79

Table 10
Performance of the algorithm CGR^p with 2-staged patterns

Instance	Solution of CGR^p	LB	Difference from LB (%)	Time (sec)	Columns generated	Solution of FFDHR	Improvement over FFDHR (%)
gcut1dr	291	291.0	0.000	0.05	26	291	0.00
gcut2dr	283	282.0	0.355	3.69	359	314	9.87
gcut3dr	317	316.0	0.316	4.35	1023	347	8.65
gcut4dr	837	836.0	0.120	9.47	1523	846	1.06
gcut5dr	175	175.0	0.000	0.09	45	198	11.62
gcut6dr	302	302.0	0.000	0.45	166	371	18.60
gcut7dr	543	542.0	0.185	0.72	193	623	12.84
gcut8dr	650	650.0	0.000	6.85	630	734	11.44
gcut9dr	126	125.0	0.800	0.10	61	143	11.89
gcut10dr	271	270.0	0.370	0.45	177	301	9.97
gcut11dr	300	299.0	0.334	8.32	677	342	12.28
gcut12dr	602	601.0	0.166	24.55	1207	696	13.51

Table 11
Performance of the algorithm CGR^p with 4-staged patterns

Instance	Solution of CGR^p	LB	Difference from LB (%)	Time (sec)	Columns Generated	Solution of FFDHR	Improvement over FFDHR (%)
gcut1dr	291	291.0	0.000	0.05	26	291	0.00
gcut2dr	283	282.0	0.355	2.22	274	314	9.87
gcut3dr	314	313.0	0.319	6.85	1103	347	9.51
gcut4dr	836	836.0	0.000	12.81	1446	846	1.18
gcut5dr	175	174.0	0.575	0.14	65	198	11.62
gcut6dr	302	301.0	0.332	0.74	230	371	18.60
gcut7dr	542	542.0	0.000	2.46	568	623	13.00
gcut8dr	651	650.0	0.154	18.35	1159	734	11.31
gcut9dr	123	122.0	0.820	0.11	58	143	13.99
gcut10dr	270	270.0	0.000	0.44	109	301	10.30
gcut11dr	299	298.0	0.336	20.08	996	342	12.57
gcut12dr	602	601.0	0.166	47.96	1535	696	13.51

For the 2-staged cutting, the algorithm obtained optimum solutions for all instances, except for two of them (on the average, the difference from LB was 0.036%). When compared to the solution of M-HFF, the improvement was 5.93% on the average. This is a great improvement, since M-HFF is also restricted to 2-staged patterns. The improvement of the 4-staged case over M-HFF was, on the average, 8.89%.

We also tested the algorithm CGR^p with stages on the instances *gcut1dr*,…,*gcut12dr*. See Tables 10 and 11. In the 2-staged case, the difference between the solution found and the lower bound was 0.22%, on the average; and the improvement over the FFDHR algorithm was around 10%. These numbers are very close to the ones we obtained for the 4-staged version.

4. The 2CS problem with bins of different sizes

In this section we adapt the algorithm CG for the 2CSV problem. Let $I = (W, H, V, w, h, d)$ be an instance of the 2CSV, where $W = (W_1, \ldots, W_b)$, $H = (H_1, \ldots, H_b)$ and $V = (V_1, \ldots, V_b)$ are lists of size b indicating the height, width, and value of each bin type i, $1 \leqslant i \leqslant b$. We can also represent each pattern j of the instance I as a vector p_j, whose ith entry indicates the number of times item i occurs in this pattern. The 2CSV problem consists then in deciding how many times each pattern has to be used to meet the demands and minimize the total value of the bins that are used. Let n be the number of all possible patterns for I, and let P denote an $m \times n$ matrix whose columns are the patterns $p_1, \ldots, p_n$. If we denote by d the vector of the demands, then the following is an ILP formulation for the 2CSV problem: minimize $\sum_{j=1}^{n} C_j x_j$ subject to $Px = d$ and $x_j \geqslant 0$ and x_j

integer for $j = 1, \dots, n$. The variable x_j indicates how many times pattern j is selected and C_j is the value of the bin type used in pattern j (note that each C_j will correspond to some V_i). The corresponding relaxed LP is the following:

$$\begin{aligned} &\text{minimize} \quad C_1x_1 + \cdots + C_nx_n \\ &\text{subject to} \quad Px = d \\ &\qquad\qquad\quad x_j \geqslant 0 \quad j = 1, \dots, n. \end{aligned} \tag{3}$$

In this case, we can also use algorithms DP and SDP to produce guillotine patterns. Moreover, if each item i has value y_i and occurs z_i times in a pattern j, produced by DP or SDP, then $\sum_{i=1}^{m} y_i z_i$ is maximum (under the pattern restrictions). This is exactly what we need to generate new columns, but in this case a column j enters the basis if $\sum_{i=1}^{m} y_i z_i > C_j$.

We describe in the sequel the algorithm SimplexCG$_3$ that solves (3) (Algorithm 4.1). In this algorithm, we have a vector f of size m that indicates the bin associated with each column of the matrix B. This way, we can reconstruct a solution considering the vector f, and the entries of B, *guillotine* and *position*. We implemented the algorithm SimplexCG$_3$ with subroutines DP and SDP to solve the RK problem.

Algorithm 4.1 SimplexCG$_3$

```
Input: An instance $I = (W, H, V, w, h, d)$ of the 2CSV problem.
Output: An optimal solution for (3), where the columns of $P$ are the patterns for $I$.
Subroutine: An algorithm $\mathcal{R}$ for the RK problem.
1 Let $f$ be a vector of size $m$, where $f_i$ is the smallest index $j$ such that $w_i \leqslant W_j$ and $h_i \leqslant H_j$.
2 Let $x = d$ and $B$ be the identity matrix of order $m$.
3 Solve $y^T B = C_B^T$. ($C_B$ is the vector $C = (C_1, \dots, C_n)$ restricted to the columns of $B$.)
4 For $i = 1$ to $b$ do
  4.1 Generate a new column $z$ executing the algorithm $\mathcal{R}$ with parameters $W_i$, $H_i$, $w$, $h$, $y$.
  4.2 If $y^T z > C_i$, go to step 6.
5 Return $B$, $f$ and $x$ and halt ($x$ corresponds to the columns of $B$).
6 Solve $Bw = z$.
7 Let $t = \min\left(\frac{x_j}{w_j} | 1 \leqslant j \leqslant m, w_j > 0\right)$.
8 Let $s = \min\left(j | 1 \leqslant j \leqslant m, \frac{x_j}{w_j} = t\right)$.
9 Let $f_j = i$
10 For $i = 1$ to $m$ do
  10.1 $B_{i,s} = z_i$.
  10.2 If $i = s$ then $x_i = t$ else $x_i = x_i - w_i t$.
11 Go to step 3.
```

We describe now the algorithm CGV (Algorithm 4.2) that solves the 2CSV problem using the algorithm SimplexCG$_3$. This algorithm is very similar to the algorithm CG of Section 3, and therefore we omit the details.

Algorithm 4.2 CGV

```
Input: An instance $I = (W, H, V, w, h, d)$ of 2CSV.
Output: A solution for $I$.
1 Execute the algorithm SimplexCG$_3$ with parameters $B$, $w$, $h$, $d$ obtaining $B$, $b$ and $x$.
2 For $i = 1$ to $m$ do $x_i^* = \lfloor x_i \rfloor$.
3 If $x_i^* > 0$ for some $i$, $1 \leqslant i \leqslant m$, then
  3.1 Return $B$, $b$ and $x_1^*, \dots, x_m^*$ (but do not halt).
  3.2 For $i = 1$ to $m$ do
    3.2.1 For $j = 1$ to $m$ do $d_i = d_i - B_{i,j} x_j^*$.
  3.3 Let $m' = 0$, $h' = ()$, $w' = ()$ and $d' = ()$.
  3.4 For $i = 1$ to $m$ do
    3.4.1 If $d_i > 0$ then $m' = m' + 1$, $w' = w' \| (w_i)$, $h' = h' \| (h_i)$ and $d' = d' \| (d_i)$.
  3.5 If $m' = 0$ then halt.
  3.6 Let $m = m'$, $w = w'$, $h = h'$, $d = d'$ and go to step 1.
4 Let $V^* = \min\left(\frac{V_i}{H_i W_i} | i = 1, \dots, b\right)$ and $j = \min\left(i | \frac{V_i}{H_i W_i} = V^*\right)$.
5 Return the solution of algorithm M-HFF executed with parameters $W_j$, $H_j$, $w$, $h$, $d$.
```

We also implemented variants of the algorithm CGV, when we may have orthogonal rotations (CGVR), staged patterns, and when the residual instance is solved with a *perturbation* method (CGV^p and CGVR^p). In the latter case, to generate a pattern we use a bin for which the fraction $\frac{V_i}{H_i W_i}$ (for $i = 1, \ldots, b$) attains the minimum value.

4.1. Computational results for the 2CSV problem

We have tested the algorithms CGV and CGV^p with the instances *gcut1d*, ..., *gcut12d*, defining three different bins. For each bin in the original instances, we define two others. Given an instance, let (W, H) be the bin dimensions of this instance. In our modified instances, one bin has dimensions $(1.2W, 0.8H)$ and the other has dimensions $(1.1W, 0.9H)$. The value of each bin corresponds to its area.

The algorithm CGV^p found solutions that are better than those obtained by the algorithm CGV, but on the average the time it spent was 58% greater.

Owing to space limitations, we show only the tests with the algorithm CGV^p, see Table 12.

For the k-staged version of he 2CSV problem, we present tests for the algorithm CGV^p with $k = 2, 4$ (see Tables 13 and 14). For $k = 2$ (resp. 4), on the average, the difference between the solution obtained by the algorithm and the lower bound was 0.50% (resp. 0.44%).

When orthogonal rotations are allowed in the 2CSV problem, we note that it becomes harder to solve the instances. The time spent by the algorithm CGVR (CGVR^p) was 81.81 (136.40) seconds on the average, while the time spent for the oriented version algorithms CGV (CGV^p) was 13.85 (21.83) seconds. Table 15 shows the results obtained by the algorithm CGV^p.

Table 12
Performance of the algorithm CGV^p

Instance	Solution of CGV^p	LB	Difference from LB (%)	Time (sec)	Columns generated
gcut1d	14,880,000	14822812.5	0.386	0.40	309
gcut2d	15,733,125	15673933.2	0.378	5.31	2771
gcut3d	19,930,000	19769831.3	0.810	33.16	15,148
gcut4d	46,346,250	46257603.4	0.192	97.02	25,871
gcut5d	41,737,500	41517500.0	0.530	0.63	499
gcut6d	74,187,500	73967812.5	0.297	2.73	1505
gcut7d	122,735,000	122295271.7	0.360	6.50	3273
gcut8d	155,602,500	155221710.8	0.245	54.85	11,933
gcut9d	129,360,000	128389230.8	0.756	1.30	1038
gcut10d	254,130,000	252565036.2	0.620	2.49	1641
gcut11d	295,250,000	292879166.7	0.809	19.59	7147
gcut12d	602,240,000	599851250.0	0.398	38.02	7392

Table 13
Performance of the algorithm CGV^p with 2-staged patterns

Instance	Solution of CGV^p	LB	Difference from LB (%)	Time (sec)	Columns generated
gcut1d	14,880,000	14822812.5	0.386	0.58	397
gcut2d	16,820,625	16740781.3	0.477	1.31	492
gcut3d	20,267,500	20149803.6	0.584	21.83	7877
gcut4d	46,591,875	46523511.2	0.147	60.56	11,569
gcut5d	42,022,500	41667500.0	0.852	0.17	110
gcut6d	78,167,500	77621562.5	0.703	0.96	539
gcut7d	124,257,500	123946562.5	0.251	2.90	1316
gcut8d	161,575,000	161074884.1	0.310	23.67	3958
gcut9d	131,830,000	130802500.0	0.786	0.12	86
gcut10d	262,470,000	260444166.7	0.778	0.81	434
gcut11d	304,440,000	303137516.6	0.430	18.58	6926
gcut12d	611,230,000	609519416.7	0.281	36.65	5452

Table 14
Performance of the algorithm CGV^p with 4-staged patterns

Instance	Solution of CGV^p	LB	Difference from LB (%)	Time (sec)	Columns generated
gcut1d	14,880,000	14822812.5	0.386	0.44	307
gcut2d	15,730,625	15673933.2	0.362	8.03	3163
gcut3d	19,864,375	19769831.3	0.478	40.23	12,327
gcut4d	46,343,750	46257603.4	0.186	125.23	18,410
gcut5d	41,737,500	41517500.0	0.530	0.68	489
gcut6d	74,187,500	73967812.5	0.297	2.27	1005
gcut7d	122,745,000	122295271.7	0.368	9.09	3715
gcut8d	155,832,500	155221710.8	0.393	117.61	17,864
gcut9d	129,360,000	128389230.8	0.756	1.01	727
gcut10d	254,130,000	252565036.2	0.620	2.76	1649
gcut11d	294,200,000	292879166.7	0.451	33.78	8413
gcut12d	602,360,000	599851250.0	0.418	68.63	7886

Table 15
Performance of the algorithm CGVR^p

Instance	Solution of CGVR^p	LB	Difference from LB (%)	Time (sec)	Columns generated
gcut1dr	13,823,750	13790625.0	0.240	0.62	402
gcut2dr	15,160,625	15083409.1	0.512	22.91	2714
gcut3dr	19,241,875	19118423.5	0.646	41.02	11,627
gcut4dr	44,663,125	44575105.3	0.197	147.12	26,535
gcut5dr	38,890,000	38454765.6	1.132	4.28	2175
gcut6dr	70,192,500	69599732.1	0.852	7.15	2646
gcut7dr	114,867,500	114503487.9	0.318	44.15	12,951
gcut8dr	152,262,500	151462312.9	0.528	529.98	47,421
gcut9dr	119,730,000	118806666.7	0.777	1.97	1093
gcut10dr	248,620,000	246552500.0	0.839	6.49	3379
gcut11dr	283,780,000	281713516.6	0.734	209.55	19,249
gcut12dr	561,680,000	559820015.8	0.332	621.60	32,690

Table 16
Performance of the algorithm CGVR^p with 2-staged patterns

Instance	Solution of CGVR^p	LB	Difference from LB (%)	Time (sec)	Columns generated
gcut1dr	13,908,750	13828125.0	0.583	0.67	416
gcut2dr	15,474,375	15432371.3	0.272	37.05	4616
gcut3dr	19,436,875	19310805.3	0.653	45.33	12,159
gcut4dr	44,905,000	44767392.4	0.307	166.68	21,902
gcut5dr	40,382,500	40087187.5	0.737	0.74	341
gcut6dr	71,162,500	70839625.0	0.456	5.49	2411
gcut7dr	115,312,500	114817716.3	0.431	56.78	13,326
gcut8dr	153,410,000	152634892.3	0.508	394.05	28,128
gcut9dr	121,040,000	119568000.0	1.231	1.14	756
gcut10dr	249,260,000	247872857.1	0.560	5.68	1545
gcut11dr	289,430,000	286973906.4	0.856	290.64	23,447
gcut12dr	564,650,000	562898801.3	0.311	690.59	28,565

For the staged version, the tests for the instances *gcut1dr*, ..., *gcut12dr* show that they are harder to be solved. We note that the large instances require several minutes to be solved; see Tables 16 and 17.

Table 17
Performance of the algorithm CGVR^p with 4-staged patterns

Instance	Solution of CGVR^p	LB	Difference from LB (%)	Time (sec)	Columns generated
gcut1dr	13,823,750	13790625.0	0.240	0.83	415
gcut2dr	15,161,875	15083409.1	0.520	46.73	3010
gcut3dr	19,181,875	19118423.5	0.332	96.76	16,255
gcut4dr	44,723,750	44575105.3	0.333	253.43	27,104
gcut5dr	38,890,000	38454765.6	1.132	4.72	1662
gcut6dr	70,192,500	69599732.1	0.852	10.45	2639
gcut7dr	114,867,500	114503487.9	0.318	70.18	12,339
gcut8dr	151,745,000	151462312.9	0.187	605.13	30,285
gcut9dr	119,730,000	118806666.7	0.777	2.73	1198
gcut10dr	248,620,000	246552500.0	0.839	9.25	3409
gcut11dr	283,560,000	281851974.2	0.606	628.87	27,379
gcut12dr	561,640,000	559820015.8	0.325	1328.03	32,554

5. The SP problem and the column generation method

The Strip Packing (SP) problem is mostly considered in the literature for the special case in which the demands are all equal to 1. Many approximation algorithms have been proposed for this problem. Coffman, et al. [21] presented the algorithms NFDH and FFDH for the oriented case with asymptotic performance bounds 2 and 1.7, respectively. Algorithms with better performance bounds were obtained by Baker et al. [4] and also by Kenyon and Rémila [34]: 5/4 and $(1+\epsilon)$. Recently, a PTAS for the SP problem with rotations was obtained by Jansen and van Stee [32].

In 2005, Seiden and Woeginger [45] presented an analysis of the quality of a k-stage guillotine strip packing versus a globally optimum packing. They showed that for $k = 2$ no algorithm can guarantee any bounded asymptotic performance ratio. When $k = 3$ (resp. $k = 4$) an asymptotic performance ratio arbitrarily close to 1.69103 (resp. 1) can be obtained.

Although some of the approximation algorithms above have bounds very close to 1, most of these results are more of theoretical relevance. Other approaches include genetic algorithms [40,10], branch-and-bound and integer linear programming models [35,37,30].

Although the column generation approach can be easily applied to the SP problem, it has been less investigated under this approach. One of the main advantages of this approach is the possibility of considering larger values of demands, as this case has many industrial applications.

Let $I = (W, H, w, h, d)$ be an instance of the SP problem. We consider that the first cut stage is done in the horizontal direction of the strip; furthermore, two subsequent cuts must be at a distance at most H. We call *H-pattern* a pattern corresponding to a packing between two subsequent horizontal cuts (that has to be at a maximum distance H).

Let $p_1, p_2, \ldots, p_n$ be the set of all possible H-patterns. Denote by H_i the height of the H-pattern p_i and let P be the matrix whose columns are the patterns $p_1, p_2, \ldots, p_n$. In this case, the following is an ILP formulation for the SP problem: minimize $\sum_{j=1}^{n} H_j x_j$ subject to $Px = d$ and $x_j \geqslant 0$ and x_j integer for $j = 1, \ldots, n$. To solve this ILP we can use the same approach we used for the problem 2CSV. In fact, we can reduce the SP problem to the 2CSV problem. For that, note that to each H-pattern of height H_i corresponds a bin with dimensions (W, H_i) and value precisely H_i.

Let $Q = \{q_1, \ldots, q_s\}$ be the set of all discretization points of the height H (this will be the maximum height of the bins). For 2-staged cutting patterns, we can consider H as the maximum height of an item, that is, $H = \max(h_1, \ldots, h_m)$. In this case, Q is the set of the heights of the items. If there are s different heights, we have H-patterns (bins) of width $W_i = W$ and height H_i, for $1 \leqslant i \leqslant s$.

The algorithm we propose to solve the SP problem, called CGS, uses basically the algorithm CGV with two modifications. First, the residual instance is solved with the algorithm FFDH. Second, every call to the algorithm SimplexCG$_3$ solves only one instance of the RK problem, consisting of a knapsack of size (W, H).

We note that, looking at the entries of V, *guillotine* and *position* produced by algorithm SDP (algorithm for the staged RK problem) we can obtain solutions for each height in Q: we just have to access positions corresponding to the dimensions (W, h_i) of these variables, for each $h_i \in Q$, $1 \leqslant i \leqslant s$. This last modification is very important, as s can be very large and solving instances of the RK problem for each of the s different bins would consume a lot of time. We did not use this idea for the 2CSV problem since it is not always better to solve only instances of RK with the largest bin dimensions.

Note that, in the 2CSV problem, the instances may consist of bins of different widths. Consider, for example, an instance consisting of x bins: one with dimensions (r, r^2), another one with dimensions (r^2, r) and the remaining ones with dimensions smaller than r, for some integer r. If we call the algorithm DP for a bin with dimensions (r^2, r^2) and assume that the number of discretization points is linearly proportional to the dimensions of the bin, then the algorithm will consume time $O(r^6)$. But if we solve for each of the bins, the algorithm will consume time $O(x\ r^5)$.

The reader should note that the first cutting phase is done automatically by the column generation algorithm by choosing the best bins in a solution. Therefore, the algorithm SDP is called with the first cutting phase in the vertical direction and one cutting phase less than the number of stages of the instance.

We implemented the algorithm CGS and its variant CGS^r (for the orthogonal rotation case) and CGS^p with a perturbed residual instance. In the algorithm CGS^p a good way to perturb the residual instance is to generate a level using the algorithm FFDH with minimum wasted area (considering the height of the level). When rotations are allowed we use an algorithm, which we denote by FFDHR2, to generate a perturbed instance. This algorithm works like the algorithm FFDH, but if an item cannot be packed in any of the existing levels then the algorithm tries to pack it in the other orientation before creating a new level.

5.1. Computational results for the SP problem

For the SP problem, we have used the instances *gcut1d,...,gcut12d* considering the maximum distance between two horizontal cuts of the strip as the width of the bin. Although the instances for the SP problem required considerably more time than the (same) instances for the 2CS problem, the corresponding times required by the latter were still small and acceptable in practice.

The results on the performance of the algorithm CGS^p for 2- and 4-staged cutting are shown in Tables 18 and 19, respectively. The lower bound corresponds to the optimal fractional solution of formulation (3).

For the 2-staged problem, all instances were solved in less than 10 seconds. On the average, the difference between the solutions found by the algorithm and the lower bound was only 0.08%. The improvement of the algorithm CGS^p over FFDH was, on the average, of 4.85%. These improvements are very significant, since algorithm FFDH also produces 2-staged solutions.

For the 4-staged problem, the difference between the solutions found by the algorithm CGS^p an the lower bound was 0.116% and the improvement over FFDH was 7.74%, on the average.

We also performed tests when orthogonal rotations are allowed. The results of the tests can be found in Tables 20 and 21. On the average, the difference between the solutions found by the algorithm CGSR^p and

Table 18
Performance of the algorithm CGS^p with 2-staged patterns

Instance	Solution of CGS^p	LB	Difference from LB (%)	Average time (sec)	Columns generated	Solution of FFDH	Improvement over FFDH (%)
gcut1d	51,604	51583.0	0.041	0.06	43	54,323	5.01
gcut2d	77,436	77369.5	0.086	0.26	141	77,436	0.00
gcut3d	80,206	80112.5	0.117	4.50	1479	83,529	3.98
gcut4d	196,480	196422.5	0.029	3.74	702	205,250	4.27
gcut5d	91,177	91177.0	0.000	0.04	29	96,693	5.70
gcut6d	168,148	167987.5	0.096	0.18	93	181,578	7.40
gcut7d	243,241	243076.0	0.068	0.65	232	259,462	6.25
gcut8d	332,924	332669.3	0.077	3.57	534	344,732	3.43
gcut9d	122,836	122532.5	0.248	0.08	66	129,706	5.30
gcut10d	272,919	272680.5	0.087	0.22	119	286,790	4.84
gcut11d	315,026	314747.5	0.088	1.50	332	338,271	6.87
gcut12d	573,806	573590.0	0.038	8.88	610	605,126	5.18

Table 19
Performance of the algorithm CGS^p with 4-staged patterns

Instance	Solution of CGS^p	LB	Difference from LB (%)	Average time (sec)	Columns generated	Solution of FFDH	Improvement over FFDH (%)
gcut1d	51,432	51332.8	0.193	0.23	178	54,323	5.32
gcut2d	77,436	77369.5	0.086	0.40	126	77,436	0.00
gcut3d	77,446	77287.0	0.206	19.80	4516	83,529	7.28
gcut4d	195,307	195249.5	0.029	9.70	1118	205,250	4.84
gcut5d	87,249	87164.4	0.097	0.11	62	96,693	9.77
gcut6d	158,137	158104.5	0.021	0.40	149	181,578	12.91
gcut7d	236,508	236412.8	0.040	1.48	314	259,462	8.85
gcut8d	310,672	310493.8	0.057	47.28	4544	344,732	9.88
gcut9d	119,861	119426.2	0.364	0.20	131	129,706	7.59
gcut10d	260,388	260259.5	0.049	0.39	132	286,790	9.21
gcut11d	305,348	304918.0	0.141	13.80	1557	338,271	9.73
gcut12d	559,159	558531.9	0.112	201.51	10,422	605,126	7.60

Table 20
Performance of the algorithm CGSR^p with 2-staged patterns

Instance	Solution of CGSR^p	LB	Difference from LB (%)	Average time (sec)	Columns generated	Solution of FFDHR2	Improvement over FFDHR2 (%)
gcut1dr	50,612	50589.0	0.045	0.10	54	54,323	6.83
gcut2dr	60,311	60192.0	0.198	1.18	347	74,744	19.31
gcut3dr	77,385	77296.3	0.115	5.88	1193	83,529	7.36
gcut4dr	175,996	175930.4	0.037	32.11	3501	191,383	8.04
gcut5dr	78,530	78370.8	0.203	0.56	235	96,530	18.65
gcut6dr	138,207	138041.0	0.120	0.97	224	181,578	23.89
gcut7dr	226,312	226163.8	0.066	3.28	531	244,742	7.53
gcut8dr	300,696	300499.3	0.065	29.54	1419	326,197	7.82
gcut9dr	119,584	119417.0	0.140	0.22	102	129,657	7.77
gcut10dr	236,531	236278.2	0.107	1.20	193	265,322	10.85
gcut11dr	286,164	285661.6	0.176	10.53	555	326,275	12.29
gcut12dr	549,751	549181.6	0.104	130.30	2908	605,126	9.15

Table 21
Performance of the algorithm CGSR^p with 4-staged patterns

Instance	Solution of CGSR^p	LB	Difference from LB	Average time (sec)	Columns generated	Solution of FFDHR2	Improvement over FFDHR2 (%)
gcut1dr	50,433	50329.0	0.207	0.47	255	54,323	7.16
gcut2dr	59,420	59124.5	0.500	10.75	2222	74,744	20.50
gcut3dr	75,396	75162.2	0.311	50.26	7261	83,529	9.74
gcut4dr	173,687	173534.3	0.088	259.84	20,740	191383	9.25
gcut5dr	74,717	74391.0	0.438	0.46	155	96,530	22.60
gcut6dr	135,952	135450.9	0.370	6.17	1332	181,578	25.13
gcut7dr	221,258	221137.5	0.054	8.19	791	244,742	9.60
gcut8dr	294,578	294188.1	0.133	764.96	23,291	326,197	9.69
gcut9dr	116,296	115927.8	0.318	0.51	164	129,657	10.30
gcut10dr	233,582	233066.7	0.221	5.24	376	265,322	11.96
gcut11dr	278,362	277230.7	0.408	206.29	5251	326,275	14.68
gcut12dr	541,998	541540.0	0.085	935.73	12,450	605,126	10.43

the lower bound was 0.11% and 0.26% respectively for the 2- and 4-staged problem. Comparing with the solutions generated by the FFDHR2 we obtain on the average an improvement of 11.6% and 13.4%, respectively, for the 2- and 4-staged problem.

6. Concluding remarks

In this paper we presented algorithms for the RK, 2CS, 2CSV and SP problems and their variants RK^r, 2CS^r, 2CSV^r and SP^r, where orthogonal rotations of the items are allowed.

For the RK problem we presented the (exact) pseudo-polynomial algorithms DP and SDP (the latter is for k-staged patterns). These algorithms can either use the algorithm DEE or DDP to generate the discretization points. We have also shown that these algorithms can be implemented to run in polynomial time when the items are not so small compared to the size of the bin. In this case the algorithms DP and SDP also run in polynomial time. We have also mentioned how to use DP and SDP to solve the problem RK^r.

We presented column generation based algorithms to solve the 2CS, 2CSV and SP problems. These algorithms use, as subroutines, the algorithms DP and SDP to generate the columns. We propose variants of the column generation algorithms that solve in different ways the residual instances.

For the 2CS and 2CSV problems, the first algorithm uses the algorithm M-HFF to solve the last residual instance and the second uses a perturbation strategy.

The algorithm CG combines different techniques: Simplex method with column generation, an exact algorithm for the discretization points, and an approximation algorithm (M-HFF) for the last residual instance. An approach of this nature has shown to be promising, and has been used to tackle the one-dimensional cutting stock problem [47,17].

The algorithm CG^p is a variant of CG, in which we use an idea that consists in perturbing the residual instances. We have also designed the algorithms CGR and CGR^p for the problems in which orthogonal rotations are allowed. The algorithm CGR uses as a subroutine (the algorithm FFDHR) that we have designed. The same ideas are used in the algorithms for the 2CSV problem.

The algorithm for the SP problem was obtained adapting the algorithm for the 2CSV problem. We have used the same strategy used in the algorithms for the 2CS and 2CSV problems. The residual instances were solved with an approximation algorithm (FFDH) or another algorithm we proposed (called FFDHR2) when rotations are allowed.

The column generation algorithms run in polynomial time, on the average, when the items are not so small compared to the size of the bin and when no perturbation is performed (under the assumption that the Simplex method runs in polynomial time on the average). The computational results with these algorithms were very satisfactory: optimal or quasi-optimal solutions were found for the instances we have considered.

For almost all instances tested, the algorithms that use a perturbation method found solutions of a slightly better quality than CG (respectively, CGR) at the cost of a slight increase in the running time.

A natural development of our work would be to adapt the approach used in the algorithm CG for the version with arbitrary orthogonal cutting patterns (the cuts need not be guillotine). One can find an initial solution using homogeneous patterns; the columns can be generated using any of the algorithms that have appeared in the literature for the two-dimensional cutting stock problem with value [6,3]. To solve the last residual instance one can use approximation algorithms [16,11,34].

One can also use column generation for the variant of 2CS in which the quantity of items in each bin is bounded (a variant proposed by Christofides and Whitlock [15]). Each new column can be generated with any of the known algorithms for the restricted two-dimensional cutting stock problem with value [15,41], and the last residual instance can be solved with the algorithm M-HFF. This restricted version with guillotine cut requirement can also be solved using the ideas we have just described: the homogeneous patterns and the patterns produced by M-HFF can be obtained with guillotine cuts, and the columns can be generated with the algorithm of Cung et al. [24].

As a final remark we mention that we did not use heuristics to solve the column generation step: for all instances we always found optimal solutions for the relaxed LP. These optimal fractional solutions yielded excellent lower bounds for the optimal solutions.

We performed many tests and compared the solutions obtained for the different variants of the problems. We noted the average percentage of increase in computational time and decrease of space occupation when we considered 2-, 3- and 4-staged patterns, as well as when rotations were considered. It should be noted that very few papers consider 4-staged patterns. Finally, we observe that for all tests performed, the algorithms we implemented found optimal or quasi-optimal solutions in a reasonable amount of time, showing that they may be useful for practical purposes.

Acknowledgement

We thank the referees for the suggestions that improved the presentation of this paper.

References

[1] Ilan Adler, Nimrod Megiddo, Michael J. Todd, New results on the average behavior of simplex algorithms, Bulletin of the American Mathematical Society (NS) 11 (2) (1984) 378–382.

[2] Ramon Alvarez-Valdes, Antonio Parajon, Jose M. Tamarit, A computational study of LP-based heuristic algorithms for two-dimensional guillotine cutting stock problems, OR Spektrum 24 (2) (2002) 179–192.

[3] M. Arenales, R. Morábito, An AND/OR-graph approach to the solution of two-dimensional non-guillotine cutting problems, European Journal of Operational Research 84 (1995) 599–617.

[4] B.S. Baker, D.J. Brown, H.P. Katseff, A $\frac{5}{4}$ algorithm for two-dimensional packing, Journal of Algorithms 2 (1981) 348–368.

[5] J.E. Beasley, Algorithms for unconstrained two-dimensional guillotine cutting, Journal of the Operational Research Society 36 (4) (1985) 297–306.

[6] J.E. Beasley, An exact two-dimensional non-guillotine cutting tree search procedure, Operations Research 33 (1) (1985) 49–64.

[7] J.E. Beasley, OR-Library: Distributing test problems by electronic mail, Journal of the Operational Research Society 41 (11) (1990) 1069–1072.

[8] G. Belov, G. Scheithauer, Models with Variable Strip Widths for Two-dimensional Two-stage cutting, www.math.tu-dresden.de/~capad/PAPERS/03-varwidth.pdf, 2003.

[9] Karl-Heinz Borgwardt, Probabilistic analysis of the simplex method, in: Brunswick, 1988 (Ed.), Mathematical Developments arising from Linear Programming, Contemp. Math., vol. 114, pp. 21–34, American Mathematical Society, Providence, RI, 1990.

[10] Andreas Bortfeldt, A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces, European Journal of Operational Research 172 (2006) 814–837.

[11] A. Caprara, Packing 2-dimensional bins in harmony, in: Proceedings of the 43rd Symposium on Foundations of Computer Science, 2002, pp. 490–499.

[12] A. Caprara, A. Lodi, M. Monaci, An approximation scheme for the two-stage, two-dimensional bin packing problem, in: A.S. Schulz, W.J. Cook (Eds.), Proceedings of the Ninth Conference on Integer Programming and Combinatorial Optimization (IPCO'02), Springer-Verlag, 2002, pp. 320–334.

[13] A. Caprara, A. Lodi, M. Monaci, Fast approximation schemes for two-stage, two-dimensional bin packing, Mathematics of Operations Research 30 (1) (2005) 150–172.

[14] A. Caprara, M. Monaci, On the two-dimensional knapsack problem, Operations Research Letters 32 (2004) 5–14.

[15] N. Christofides, C. Whitlock, An algorithm for two dimensional cutting problems, Operations Research 25 (1977) 30–44.

[16] F.R.K. Chung, M.R. Garey, D.S. Johnson, On packing two-dimensional bins, SIAM Journal of Algebraic and Discrete Methods 3 (1982) 66–76.

[17] G.F. Cintra, Algoritmos hı́bridos para o problema de corte unidimensional, in: XXV Conferência Latinoamericana de Informática, Assunção, 1999.

[18] G.F. Cintra, Algoritmos para problemas de corte de guilhotina bidimensional, PhD thesis, Instituto de Matemática e Estatı́stica, São Paulo, 2004.

[19] G.F. Cintra, F.K. Miyazawa, Y. Wakabayashi, E.C. Xavier, A note on the approximability of cutting stock problems, European Journal on Operations Research 183 (3) (2007) 1328–1332.

[20] G.F. Cintra, Y. Wakabayashi, Dynamic programming and column generation based approaches for two-dimensional guillotine cutting problems, in: Proceedings of WEA 2004: Workshop on Efficient and Experimental Algorithms, Lecture Notes in Computer Science, vol. 3059, 2004, pp. 175–190.

[21] E.G. Coffman Jr., M.R. Garey, D.S. Johnson, R.E. Tarjan, Performance bounds for level oriented two-dimensional packing algorithms, SIAM Journal on Computing 9 (1980) 808–826.

[22] COIN-OR Linear Program Solver, An Open Source code for Solving Linear Programming Problems, http://www.coin-or.org/Clp/index.html.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, second ed., MIT Press, Cambridge, MA, 2001.

[24] Van-Dat Cung, Mhand Hifi, Bertrand Le Cun, Constrained two-dimensional cutting stock problems a best-first branch-and-bound algorithm, Int. Trans. Oper. Res. 7 (3) (2000) 185–210.

[25] L. Epstein, Two dimensional packing: The power of rotation, in: Proceedings of the 28th International Symposium of Mathematical Foundations of Computer Science, Lecture Notes on Computer Science – LNCS, vol. 2747, Springer–Verlag, 2003, pp. 398–407.

[26] S.P. Fekete, J. Schepers, J.C. van der Veen, An exact algorithm for higher-dimensional orthogonal packing, Operations Research 55 (3) (2007) 569–587.

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

[28] P. Gilmore, R. Gomory, A linear programming approach to the cutting stock problem – Part II, Operations Research 11 (1963) 863–888.
[29] P. Gilmore, R. Gomory, Multistage cutting stock problems of two and more dimensions, Operations Research 13 (1965) 94–120.
[30] M. Hifi, Exact algorithms for the guillotine strip cutting/packing problem, Computers & Operations Research 25 (11) (1998) 925–940.
[31] J.C. Herz, A recursive computational procedure for two-dimensional stock-cutting, IBM Journal of Research and Development (1972) 462–469.
[32] K. Jansen, R. van Stee, On strip packing with rotations, in: Proceedings of the ACM Symposium on Theory of Computing, 2005, pp. 755–761.
[33] K. Jansen, G. Zhang, On rectangle packing: maximizing benefits, in: Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, 2004, pp. 204–213.
[34] C. Kenyon, E. Rémila, A near-optimal solution to a two-dimensional cutting stock problem, Mathematics of Operations Research 25 (2000) 645–656.
[35] A. Lodi, S. Martello, D. Vigo, Models and bounds for two-dimensional level packing problems, Journal of Combinatorial Optimization 8 (2004) 363–379.
[36] A. Lodi, M. Monaci, Integer linear programming models for 2-staged two-dimensional knapsack problem, Mathematical Programming 94 (2003) 257–278.
[37] S. Martello, M. Monaci, D. Vigo, An exact approach to the strip-packing problem, INFORMS Journal on Computing 15 (3) (2003) 310–319.
[38] F.K. Miyazawa, Y. Wakabayashi, Packing problems with orthogonal rotations, in: Proceedings of Latin American Theoretical INformatics, Lecture Notes in Computer Science, vol. 2976, Springer-Verlag, Buenos Aires, Argentina, 2004, pp. 359–368.
[39] R. Morábito, M. Arenales, V.F. Arcaro, An and-or-graph approach for two-dimensional cutting problems, European Journal of Operational Research 58 (1992) 263–271.
[40] E.A. Mukhacheva, A.S. Mukhacheva, The rectangular packing problem: Local optimum search methods based on block structures, Automation and Remote Control 65 (2) (2004) 101–112.
[41] J.F. Oliveira, J.S. Ferreira, An improved version of Wang's algorithm for two-dimensional cutting problems, European Journal of Operational Research 44 (1990) 256–266.
[42] J. Puchinger, G.R. Raidl, An evolutionary algorithm for column generation in integer programming: an effective approach for 2d bin packing, in: Proceedings of Parallel Problem Solving from Nature – PPSN VIIILNCS, vol. 3242, Springer-Verlag, 2004, pp. 642–651.
[43] J. Puchinger, G.R. Raidl, Models and algorithms for three-stage two-dimensional bin packing, European Journal of Operational Research 183 (3) (2007) 1304–1327.
[44] J.R. Riehme, G. Scheithauer, J. Terno, The solution of two-stage guillotine cutting stock problems having extremely varying order demands, European Journal of Operational Research 91 (1996) 543–552.
[45] S.S. Seiden, G.J. Woeginger, The two-dimensional cutting stock problem revisited, Mathematical Programming 102 (3) (2005) 519–530.
[46] F. Vanderbeck, A nested decomposition approach to a 3-stage 2-dimensional cutting stock problem, Management Science 47 (2001) 864–879.
[47] G. Wäscher, T. Gau, Heuristics for the integer one-dimensional cutting stock problem: A computational study, OR Spektrum 18 (1996) 131–144.
[48] G. Wäscher, H. Haussner, H. Schumann, An improved typology of cutting and packing problems, European Journal of Operational Research 183 (3) (2007) 1109–1130.