Comput Optim Appl (2009) 42: 303–326
DOI 10.1007/s10589-007-9081-5

Approximate and exact algorithms for the double-constrained two-dimensional guillotine cutting stock problem

M. Hifi · R. M'Hallah · T. Saadi

Received: 20 January 2006 / Revised: 18 July 2006 / Published online: 30 October 2007

Abstract In this paper, we propose approximate and exact algorithms for the double constrained two-dimensional guillotine cutting stock problem (DCTDC). The approximate algorithm is a two-stage procedure. The first stage attempts to produce a starting feasible solution to DCTDC by solving a single constrained two dimensional cutting problem, CDTC. If the solution to CTDC is not feasible to DCTDC, the second stage is used to eliminate non-feasibility. The exact algorithm is a branch-and-bound that uses efficient lower and upper bounding schemes. It starts with a lower bound reached by the approximate two-stage algorithm. At each internal node of the branching tree, a tailored upper bound is obtained by solving (relaxed) knapsack problems. To speed up the branch and bound, we implement, in addition to ordered data structures of lists, symmetry, duplicate, and non-feasibility detection strategies which fathom some unnecessary branches. We evaluate the performance of the algorithm on different problem instances which can become benchmark problems for the cutting and packing literature.

Keywords Combinatorial optimization · Cutting problems · Dynamic programming · Single constrained knapsack problem

M. Hifi (✉)
LaRIA, Université de Picardie Jules Verne, 5 rue du Moulin Neuf, 80000 Amiens, France
e-mail: hifi@u-picardie.fr

R. M'Hallah
Department of Statistics and Operations Research, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait
e-mail: mhallah@kuc01.kuniv.edu.kw

M. Hifi · T. Saadi
CERMSEM, MSE, Université Paris, 1 Panthéon-Sorbonne, 106-112, bd de l'Hôpital, 75013 Paris, France

1 Introduction

Several real life industrial applications require the allocation of a set of rectangular items (or pieces) to a larger rectangular standardized stock unit. For instance, in the glass, plastic or metal industries, rectangular components have to be cut from a large sheet of material. Similarly, when filling the pages of a newspaper, the editor has to arrange articles and advertisements, presented as rectangular areas, on pages of fixed dimensions. Generally, industrial applications require cutting or packing the largest number of items/pieces into a rectangular unit as to minimize the waste of the rectangular stock unit. This cutting (or packing) problem is a difficult combinatorial optimization problem, known in the literature as the Two-Dimensional Cutting stock problem (TDC).

The TDC problem is one of the most interesting cutting and packing problems. Cutting and Packing problems, referred to as CP in Dyckhoff [14], constitute a family of natural combinatorial optimization problems, admitted in numerous real-world applications such as computer science, industrial engineering, logistics, manufacturing and production processes. In industry, the quantities produced of each item are generally constrained. For instance, metal sheets, plastics, glass, and photographic films production is always subject to demand constraints. These constraints make the TDC problem more complex and harder to solve.

In this paper, we solve the *Double Constrained TDC* guillotine problem (DCTDC), which consists of cutting, from a large rectangle S of dimensions $L \times W$, n small rectangles (called items or pieces) of values (weights or profits) c_i and dimensions $l_i \times w_i$, $\forall i \in I = \{1, \ldots, n\}$, and $w_1 \geq w_2 \geq \cdots \geq w_n$. Associated to each piece type $i \in I$ is a couple of non negative integer numbers (a_i, b_i), $0 \leq a_i \leq b_i$ where a_i denotes the lower demand value imposed on piece type i, b_i the upper demand value, and $\sum_{i \in I} a_i > 0$. In addition,

1. all pieces have a *fixed orientation*, i.e., a piece of length l and width w is different from a piece of length w and width l (when $l \neq w$);
2. all applied cuts are of *guillotine* type, i.e., cuts start from one edge of the stock sheet and run parallel to the other two edges as illustrated in Fig. 1(a);
3. a feasible cutting pattern has at *least* a_i and at *most* b_i pieces of type i; and
4. all parameters L, W, l_i, and w_i, $i \in I$, are nonnegative integers.

The n-dimensional vector of non-negative integer numbers $(d_1, \ldots, d_n)$ is a *cutting pattern* to DCTDC if it is possible to pack d_i pieces of type $i \in I$ in the large rectangle S without overlap, and to cut them using guillotine cuts. This cutting pattern is feasible if $\forall i \in I$, $a_i \leq d_i \leq b_i$. The DCTDC problem is *unweighted* if $c_i = l_i \times w_i$, $i \in I$, and *weighted* otherwise. Formally, the DCTDC problem can be stated as

$$(DCTDC) \begin{cases} \max & \displaystyle\sum_{i \in I} c_i d_i, \\ \text{subject to} & (d_1, \ldots, d_n) \text{ a } \textit{cutting pattern} \\ & a_i \leq d_i \leq b_i, \ i \in I. \end{cases} \tag{1}$$

When all $a_i = 0$, $i \in I$, the problem becomes a single constrained TDC, CTDC. When, in addition, all $b_i = \infty$, $i \in I$, the CTDC reduces to the unconstrained TDC problem, UTDC.

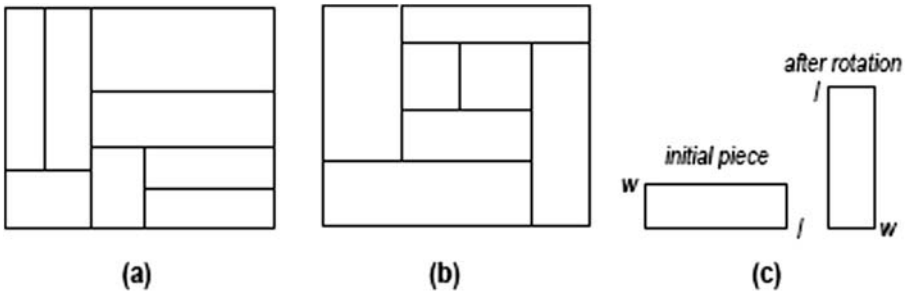

Fig. 1 **a** A guillotine cutting pattern, **b** a non-guillotine solution, and **c** a 90° rotation of a piece of dimensions $l \times w$

In this paper, we propose an exact algorithm to solve the double constrained two dimensional guillotine cutting problem. The algorithm is an extension of the approach proposed in Cung and Hifi [11] and Cung et al. [13], with several modifications. It is a bottom-up branch and bound using a best-first search strategy. The success of this algorithm is mainly due to the designed bounding scheme, to the symmetry, duplicate, and non-feasibility detection strategies that reduce the size of the branching tree, and to the adopted data structure.

This paper is organized as follows. In Sect. 2, we briefly discuss some of the sequential approaches used to solve the TDC problem, and indicate the state of the art for the DCTDC problem. In Sect. 3, we propose an approach that identifies a feasible initial solution to DCTDC. In Sect. 4, we describe the proposed exact branch and bound algorithm, explain how to obtain upper bounds, and provide some branch and bound strategies whose implementation enhances the algorithm. In Sect. 5, we assess the performance of the proposed algorithm on a set of problem instances which can eventually become benchmark results for the literature. Finally, in Sect. 6, we summarize the contributions of this paper and their possible extensions.

2 Literature review

Because of its importance and despite its NP-hardness, the TDC problem has been widely studied in the literature [4–6, 15, 18, 30, 31]. Most of the contributions dealing with the TDC problem focused on the fixed orientation TDC where the items have a fixed orientation. The fixed orientation case reflects a number of practical contexts, such as the cutting of corrugated or decorated material, newspapers paging, etc. Variants, reflecting other real-world applications—such as the 90° rotation (e.g., Fig. 1(c)), the guillotine cutting, the k staged guillotine TDC problem (see Gilmore and Gomory [19], Hifi and M'Hallah [23], Lodi and Monaci [28], Belov and Scheithauer [3], and Morabito and Arenales [29], etc.—have similarly been investigated. General surveys on cutting and packing problems can be found in Lodi et al. [27] and in the recent annotated bibliography provided by Wäescher et al. [34].

The design of a solution approach to the TDC problem depends generally on the particular framework of its application, and on the available computational resources. For instance, the UTDC problem was solved exactly using dynamic programming

(see Beasley [2], and Gilmore and Gomory [20]), search tree procedures and hybrid methods (see, Cui et al. [10], Herz [21], and Hifi and Zissimopoulos [25]), and solved approximately using heuristic approaches (see Beasley [2], Fayard et al. [17], Morabito and Arenales [29], and Wäescher et al. [34]). The CTDC problem was solved exactly by a depth-first search method (see Christofides and Whitlock [7]). It was also solved by a best-first search method (see Cung et al. [12, 13], and Viswanathan and Bagchi [32]) which is a generalization of Wang's approach (see Wang [33]) since it solves both unweighted and weighted versions of the CTDC problem.

To our knowledge, the DCTDC problem was never investigated in the literature. No set of benchmark instances is available in the literature for this problem. Herein, we solve the DCTDC problem approximately using a two-stage procedure (Sect. 3), and exactly using a branch and bound algorithm (Sect. 4).

3 An initial solution to DCTDC

Finding an initial solution to DCTDC is itself an NP complete problem. Herein, we find an initial feasible solution to DCTDC using an approximate *two-stage* procedure inspired from Hifi and M'Hallah [24]. The procedure solves CTDC, a DCTDC relaxation obtained by ignoring the lower demand constraints. If the solution happens to satisfy the relaxed constraints, then it is an *initial lower bound* used at the root node. Otherwise, feasibility is restored using a greedy procedure (GP) or using an improved greedy procedure (IGP). In the following, we detail the first stage which is a constructive procedure that identifies a solution to the NP hard CTDC problem. We then detail GP and IGP whose objective is to restore feasibility when the solution of the first stage is infeasible.

3.1 A constructive procedure

Solving the NP-hard CTDC is very complex. Thus, we approximately solve it using a constructive approach which generates strips and combines them as explained below.

1. Let r be the number of different widths of the pieces such that $\overline{w}_1 > \overline{w}_2 > \cdots > \overline{w}_r$, where $r \leq n$.
   Set $b'_i = b_i$, $d'_i = 0$, $i \in I$, and $\rho = 0$, where ρ is the number of strips in the final solution.
2. Set $j = 1$, and $W' = W$.
3. Generate a strip of width $\overline{w}_j$ by solving—as in [24]—the *Bounded Knapsack* (BK) problem:

$$(BK^{hor}_{L,\overline{w}_j}) \begin{cases} f^{hor}_{\overline{w}_j}(L) = \max \displaystyle\sum_{i \in S_{L,\overline{w}_j}} c_i d_{ij} \\ \text{subject to} \displaystyle\sum_{i \in S_{L,\overline{w}_j}} l_i d_{ij} \leq L,\ d_{ij} \leq b'_i,\ d_{ij} \in N,\ i \in S_{L,\overline{w}_j}, \end{cases}$$

   where d_{ij} denotes the number of times piece type i appears in $(L, \overline{w}_j)$ without exceeding its upper residual demand value b'_i. $S_{L,\overline{w}_j}$ is the set of piece types that

Algorithm 1. Restoring feasibility using GP

1. Let ρ be the number of strips used to construct d'.
   Let $S_\delta = \{i \in I \mid d'_i < a_i\}$, $\bar{S}_\delta = \{i \in I \mid d'_i > a_i\}$, and $n_{strip} = 0$.
2. Let $j = 1$.
3. Let $J \subset \bar{S}_\delta$ be the set of pieces contained in strip j of d' and whose lower and upper demand constraints are satisfied; that is, $J = \{i \mid i \in \bar{S}_\delta \text{ and } d'_{ij} > 0\}$.
   (a) Remove from strip j all piece types of J; update d' by setting $d'_i = d'_i - d'_{ij}$, for $i \in J$.
   (b) Move all remaining pieces of strip j to the bottom-left.
   (c) Define the obtained hole.

   For each piece type $i \in \bar{S}_\delta$, pack in the hole—using a *bottom left procedure* [1, 23] which generates only guillotine patterns—as many pieces as possible not exceeding $a_i - d'_i$ pieces.
   If none of the pieces of S_δ fit into the hole, use pieces from $\bar{S}_\delta$.

   (d) Update S_δ and $\bar{S}_\delta$.
   (e) Set $n_{strip} = n_{strip} + 1$.
4. If $S_\delta = \emptyset$, stop: the solution is feasible.
   If $S_\delta \neq \emptyset$, and $\bar{S}_\delta = \emptyset$, stop: the solution is infeasible.
   If $S_\delta \neq \emptyset$, and $\bar{S}_\delta \neq \emptyset$, and $n_{strip} > \rho$, stop: the solution is infeasible.
   If $S_\delta \neq \emptyset$, and $\bar{S}_\delta \neq \emptyset$, and $n_{strip} \leq \rho$, goto Step 3.

can fit into strip (L, w_j) and whose residual demand value is strictly positive; that is, $S_{L,\overline{w}_j} = \{i \in I \mid (l_i, w_i) \leq (L, \overline{w}_j) \text{ and } b'_i > 0\}$.
4. Set $d'_i = d'_i + d_{ij}$, $i \in I$.
5. Update the residual demand by setting $b'_i = b'_i - d_{ij}$, $i \in I$, and update the current number of strips by setting $\rho = \rho + 1$.
6. Position the current strip $(L, \overline{w}_j)$ in the containing rectangle, and reduce the current width of the containing rectangle W' by $\overline{w}_j$.
7. If $\{k \in I : w_k = \overline{w}_j,\ b'_k > 0\} = \emptyset$, then
   If $\exists i \in I$ such that $b'_i > 0$, and $W' \geq w_i$, set $j = i$ and goto Step 3.
   Else, stop.
   Else goto Step 3.

Step 3 is solved several times; generating several strips of width $\overline{w}_j$, $j \in \{1, \ldots, r\}$, where the contents of two strips of the same width may be different.

3.2 Restoring feasibility

The constructed solution $d' = (d'_1, \ldots, d'_n)$ is feasible to CTDC (the relaxed DCTDC problem) but not necessarily to DCTDC. In fact, there may exist piece types $i \in I$ whose lower demand constraints are not satisfied; i.e., $d'_i < a_i$. To overcome infeasibility, we apply the complementary greedy procedure (GP) detailed in Algorithm 1.

Step 3 of GP generates a hole by removing a number of pieces from a strip, and replaces the removed pieces by pieces whose lower demand constraints are violated.

Algorithm 2. Combining horizontal and/or vertical strips: An *Improved* GP (IGP)

Initialization
Set $Sol_H = -\infty$, $Sol_V = -\infty$, where Sol_H (resp. Sol_V) corresponds to the solution value reached by GP (cf., Sect. 3.2).
Set $Best = \max\{0, Sol_H, Sol_V\}$, where $Best$ denotes the best current solution value obtained up to now.
Improvement
$\forall k \in \{1, \ldots, r\}$, do:
Set $W_{rest} = W - w_k$ and $Stop = false$.

Repeat

1. Fill R_{y_k} using a horizontal GP.
2. Complete $R_{W_{rest}}$ using a vertical GP.
3. Set $Best = \max\{Best, Best_{(R_{y_k}, R_{W_{rest}})}\}$, where $Best_{(R_{y_k}, R_{W_{rest}})}$ denotes the solution value of $(R_{y_k}, R_{W_{rest}})$.
4. If $\exists (l_j, w_j) \subseteq (L, \omega)$, such that $\omega \in \{w_1, \ldots, w_r\}$ and $\omega \leq W_{rest}$, then set $y_k = y_k + \omega$;
   Else set $Stop = true$.

Until $Stop$.

If none of the pieces with unsatisfied demand (i.e., pieces from S_δ) fit into the hole, the other pieces (from $\bar{S}_\delta$) are used to fill the hole. Indeed, Step 3c favors pieces from S_δ. GP stops with a feasible solution when the lower demand constraints are all satisfied (i.e., $S_\delta = \emptyset$). It stops with an infeasible solution when there is no longer any piece type i whose $d'_i > a_i$ while there exists at least a piece type whose lower demand constraint is not attained (i.e., $\bar{S}_\delta = \emptyset$, and $S_\delta \neq \emptyset$). If both $\bar{S}_\delta$ and S_δ are non empty, GP iterates through Step 3 as long as the number of strips so far considered does not exceed ρ; i.e., $n_{strip} \leq \rho$. When $n_{strip} > \rho$, GP exits with an infeasible solution.

3.3 Using a discretization procedure

In order to improve the obtained solution, or overcome its infeasibility, we introduce a second stage. This stage creates a set $\mathcal{R}$ using a discretization procedure (as used in [9, 16, 17, 23]), and combines the elements of $\mathcal{R}$ using a constructive procedure. $\mathcal{R}$ is a finite set whose elements $\overline{R}$ represent a couple of two complementary sub-rectangles, where $\overline{R} = (\overline{R}_y, \overline{R}_{W-y})$, $y \in \{w_1, \ldots, w_r\}$ when a horizontal or y-cut is performed. $\overline{R} = (\overline{R}_x, \overline{R}_{L-x})$, $x \in \{l_1, \ldots, l_s\}$ represents the couple of complementary rectangles obtained when a vertical or x-cut is performed, where s is the number of distinct lengths of the piece types. Herein, we focus on horizontal cuts since applying vertical cuts is a straightforward adaptation requiring the swap of the dimensions.

When solving $BK^{hor}_{L,W}$ using dynamic programming, we obtain—in addition to the widest horizontal strip—all optimal sub-strips (L, β), where $\beta \leq W$. We apply the procedure IGP detailed in Algorithm 2 to get $\overline{R}_y = (R_y, R_{W-y}) \in \mathcal{R}$ when a y-cut is performed on the width W.

Algorithm 3. The branch and bound exact algorithm: ALGO

Open: the set of subproblems
Clist: the list of stored best subproblems
R and q: the guillotine rectangles
$g(R)$, $h'(R)$, $f'(R)$: the internal value, the upper bound of the region P and the upper bound of the subproblem containing the configuration R.
best, *Opt*: the best current feasible solution and its solution value.

`Input:` an instance of the DCTDC problem.
`Output:` an optimal solution *best* of value *Opt*.

- *Open* $:= \{R_1, R_2, \ldots, R_n\}$.
- *Clist* $:= \emptyset$. finished:=`false`.
- Compute $Upper(x_R, y_R)$ for each $x_R = 1, \ldots, L$, and $y_R = 1, \ldots, W$.
- Call IGP to produce a starting solution *best* of value $Opt(best)$.

`repeat`
choose from *Open* the element R having the highest $f'(R)$ value;
`if` $(f'(R) - Opt(best) \leq 0)$ `then` finished := `true`
`else`
`begin`
1. transfer R from *Open* to *Clist*;
2. construct all guillotine rectangles Q such that:
2.1. Q is a horizontal or vertical build of R with some rectangle R' (including R) of *Clist*;
2.2. $Q \leq (L, W)$;
2.3. $\forall q \in Q, d_k^q \leq b_k, k \in I$;
3. `for` each element $q \in Q$, `do`:
3.1. `if` $[g(q) > Opt(best)]$ and $[d_k^q \geq a_k,\ k \in I]$, `then` set $best = q$ and $Opt(best) = g(q)$;
3.2. `if` $[f'(q) > Opt(best)]$ `then` set $Open = Open \cup \{q\}$;
4. remove all elements R from *Open* having $f'(R) \leq Opt(best)$ or $h'(R) = 0$;
5. `if` $Open = \emptyset$ `then` finished := `true`;
`end`;
`until` finished;

IGP is a greedy procedure. It sets the initial solution to GP's solution which could have been obtained by horizontal and/or vertical strips. During its improvement iterations, IGP fixes a width $y \in \{w_1, \ldots, w_r\}$ (or a length $x \in \{l_1, \ldots, l_s\}$) on the stock rectangle S, fills the first sub-rectangle (L, y) (or (x, W)), and combines the horizontal and/or vertical strips using GP. Steps 1–4 are repeated till all pieces are packed or till no piece can fit into the latest created strip.

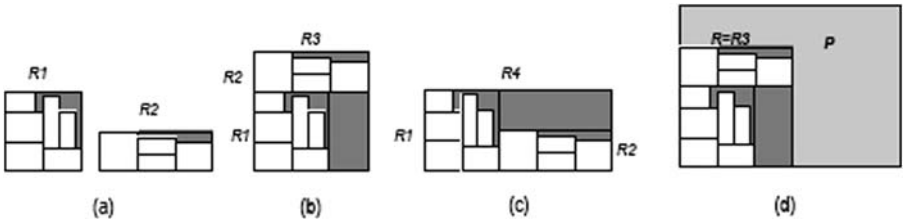

Fig. 2 **a** Two internal subrectangles R_1 and R_2, **b** a vertical build of subrectangles R_1 and R_2, **c** a horizontal build of subrectangles R_1 and R_2, and **d** $P = S \backslash R$ is the complement of the internal rectangle R in S

4 An exact method for the DCTDC

We solve DCTDC using an exact branch and bound algorithm which uses the objective function value of the feasible solution identified in Sect. 3 as a lower bound. Each node of the branch and bound tree corresponds to a feasible internal rectangle. An internal rectangle of the stock rectangle S is obtained via vertical or horizontal builds. A vertical build $\frac{R_1}{R_2}$ of two subrectangles $R_1 = x_{R_1} \times y_{R_1}$ and $R_2 = x_{R_2} \times y_{R_2}$ is the *internal* subrectangle $R_3 = \max\{x_{R_1}, x_{R_2}\} \times (y_{R_1} + y_{R_2})$ displayed in Fig. 2(b). A horizontal build $R_1|R_2$ of R_1 and R_2 is the *internal* subrectangle $R_4 = (x_{R_1} + x_{R_2}) \times \max\{y_{R_1}, y_{R_2}\}$ shown in Fig. 2(c). An internal rectangle $R = x_R \times y_R$ is feasible if

- $\forall (l_i, w_i) \subseteq R, d_i^R \leq b_i$, where d_i^R is the number of times piece type i appears in R;
- R can fit into the large stock rectangle S, that is, $(x_R, y_R) \leq (L, W)$; and
- $a_i \leq d_i^R + d_i^P \leq b_i$, $i \in I$, where as displayed in Fig. 2(d), $P = S \setminus R$, and R is positioned at the bottom left corner of S.

For each internal rectangle R, the algorithm computes an upper bound. Let $g(R)$ be the *value* of R; that is, $g(R)$ is the sum of the profits of the pieces contained in R. Let $h(R)$ be the maximum profit which could be obtained by filling the region $P = S \setminus R$ of Fig. 2(d) without violating any of the demand constraints. Then $f(R) = g(R) + h(R)$ represents an upper bound on the *maximum profit* generated by any feasible cutting pattern constrained to include R. Computing $h(R)$ is not a straightforward task; thus, we determine an estimate $h'(R)$ of $h(R)$, and deduce an estimate $f'(R)$ for the upper bound $f(R)$. Several approaches are used to derive the tightest upper bound. The algorithm branches on the most promising internal node while it fathoms infeasible nodes and nodes which can not yield an optimal solution.

The steps of the algorithm are detailed in Sect. 4.1. The computation of the upper bounds is explained in Sect. 4.2. The implementation strategies including the adopted data structure, the symmetry, the detection of duplicate and non-feasible patterns are detailed in Sect. 4.3. Finally, some remarks are provided in Sect. 4.4.

4.1 The main steps of the exact algorithm

ALGO (see Algorithm 3) uses two main lists: *Open* and *Clist*. The *Open* list initially contains n elements. Each element R_i, $i \in I$ is identical to piece type i; i.e., $(l_{R_i}, w_{R_i}) = (l_i, w_i)$. Associated to each element R_i is a set of characteristics including its value $g(R_i)$, an estimate for $h'(R_i)$, the corresponding upper bound $f'(R_i)$,

and a vector d^{R_i} of dimension n, where $d_k^{R_i}$, $k \in I$, is the number of occurrences of piece type k in the internal rectangle R_i. Initially, the algorithm sets $g(R_i) = c_i$, $d_k^{R_i} = 1$ if $k = i$ and $d_k^{R_i} = 0$, otherwise.

The *Clist* is initialized to the empty set. At each iteration, the element R with the maximal $f'(R)$ is selected from *Open* and moved to the *Clist*. A set Q of new rectangles is created by combining R with elements of *Clist* (including R itself) via horizontal and vertical builds. Only those rectangles q satisfying $(l_q, w_q) \leq (L, W)$, $d_k^q \leq b_k$, $k \in I$ (the upper demand values) are included in set Q. The current incumbent feasible solution *best* with value $Opt(best)$ is replaced by the current solution q with value $g(q)$ if (i) $d_k^q \geq a_k$, $k \in I$ and (ii) $g(q) > Opt(best)$; in this case, we set the cutting pattern *best* equal to q and the best current solution value $Opt(best)$ equal to $g(q)$. Finally, the algorithm stops when either *Open* is empty or the selected rectangle R from *Open* has a value $f'(R)$ less than or equal to the best current solution value $Opt(best)$.

4.2 Upper bounds at internal nodes

For each internal rectangle R, the algorithm computes an upper bound. We propose several ways to compute an upper bound $f'(R)$ of the optimal solution value of S when it includes the current internal rectangle $R = x_R \times y_R$ whose value is $g(R)$. The upper bound $f'(R) = g(R) + h'(R)$, where $h'(R)$ is the maximum profit that can be achieved by filling region $P = S \setminus R$. This bound is the tightest of bounds obtained by solving—as in [22, 24, 26]—an UTDC, a single bounded knapsack (SBK), and a relaxed bounded knapsack (RSBK).

4.2.1 The first upper bound

An upper bound for DCTDC is the solution value obtained by solving UTDC, a relaxed version of DCTDC where the demand constraints are dropped. In this case, $h'(R)$ is estimated by

$$\mathcal{U}_1(x_R, y_R) = \min\{F(L, W - y_R) + F(L - x_R, W), F(L, W) - g(R)\}, \quad (2)$$

where F is the solution value of the two-dimensional knapsack function of Gilmore and Gomory [20]. $\mathcal{U}_1(x_R, y_R)$ is suspected to be not very tight since the first term of the right hand side of (2) accounts for the rectangle $(L - x_R, W - y_R)$ twice.

4.2.2 The second upper bound

A second upper bound, $\mathcal{U}_2(x_R, y_R)$, is the solution value of the Single Bounded Knapsack problem $SBK_{(x_R, y_R)}$.

$$\mathcal{U}_2(x_R, y_R) = \max\left\{ \sum_{i \in S_P^R} c_i z_i, \text{ s.t. } \sum_{i \in S_P^R} (l_i w_i) z_i \leq (LW - x_R y_R), \right.$$
$$\left. z_i \leq \min\left\{ b_i, \left\lfloor \frac{LW - x_R y_R}{l_i w_i} \right\rfloor \right\}, \ z_i \in \mathbb{N}, \ i \in S_P^R \right\}.$$

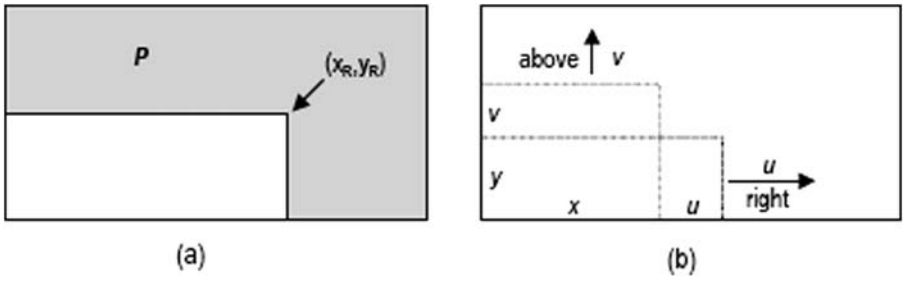

Fig. 3 **a** An internal subrectangle (x_R, y_R) placed on the bottom left of the initial rectangle, and **b** the mechanism used to compute the upper bound U_3

S_P^R is the set of pieces that can be cut from the region P, and z_i is the number of occurrences of piece type i in P. For all R, $\mathcal{U}_2(x_R, y_R)$ are readily available for any node of the tree. At the beginning of the algorithm, the largest knapsack $SBK_{(0,0)}$ is solved using dynamic programming yielding the solutions for all $SBK_{(x,y)}$, $x \leq L$ and $y \leq W$. Thus, the upper bounds for all (x_R, y_R) are available at the beginning of the algorithm.

4.2.3 Enhancing the current upper bound

We enhance the upper bounds $\mathcal{U}_1(x_R, y_R)$ and $\mathcal{U}_2(x_R, y_R)$ by (i) applying Viswanathan and Bagchi's recursive algorithm (see Viswanathan and Bagchi [32]), and (ii) solving a single bounded knapsack problem using a greedy procedure and a polynomial time procedure.

Using recursion The recursion function enumerates all possible ways that allow the internal rectangle R to be positioned at the bottom-left corner of a guillotine cutting pattern as in Fig. 3(a). To determine the boundary conditions, we apply the recursion to points which lie within the large rectangle S when we are stepping off from points to the right or on top of R, as shown in Fig. 3(b). Subsequently, for each current internal rectangle R, the solutions to the UTDC and $SBK_{(.,.)}$ problems can be combined to yield a tighter upper bound

$$\mathcal{U}_3(x_R, y_R) = \min\{\mathcal{H}(R), \mathcal{V}(R)\} \tag{3}$$

where

$$\mathcal{H}(R) = \max\{\mathcal{U}_3(x_R + u, y_R) + \min\{\mathcal{U}_1(x_R + u, y_R), \mathcal{U}_2(u \times y_R)\} : 1 \leq u \leq L - x_R\}, \tag{4}$$

$$\mathcal{V}(R) = \max\{\mathcal{U}_3(x_R, y_R + v) + \min\{\mathcal{U}_1(x_R, y_R + v), \mathcal{U}_2(x_R \times v)\} : 1 \leq v \leq W - y_R\}, \tag{5}$$

and $\mathcal{U}_3(L, W) = 0$.

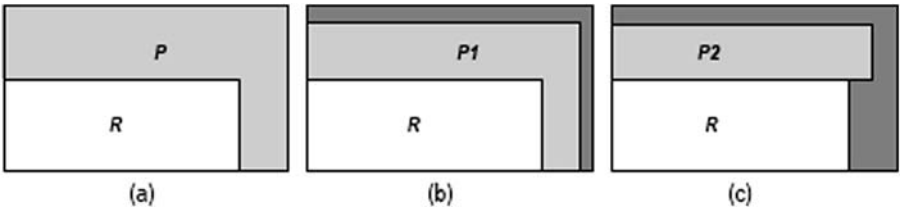

Fig. 4 **a** The initial area P to be estimated when R is placed at the bottom left corner of the stock rectangle S, **b** the normalized area $P1$, and **c** the reduced area P_2 when applying (6)

Using a single bounded knapsack problem Equation (3) chooses the smallest of $\mathcal{U}_1(x_R, y_R)$ and $\mathcal{U}_2(x_R, y_R)$ obtained respectively by UTDC and *SBK*. Instead of solving $SBK_{(x_R, y_R)}$, we can solve RSBK, a relaxed version of SBK, and use its solution value $\mathcal{U}_4(x_R, y_R)$ as an upper bound. That is,

$$\mathcal{U}_4(x_R, y_R) = \max\left\{ \sum_{i \in S_P^R} c_i z_i, \text{ s.t. } \sum_{i \in S_P^R} (l_i w_i) z_i \le T,\ a_i' \le z_i \le b_i',\ z_i \ge 0,\ i \in S_P^R \right\},$$

where the bounds on the demand for piece type $i \in I$ are relaxed to $(a_i', b_i') = (a_i - d_i^R, \min\{b_i, \lfloor \frac{T}{l_i w_i} \rfloor\})$. The value T, which depends on the pieces that can fit into P, is computed as:

$$T = \begin{cases} L(W - y_R) & \text{if } \nexists k,\ k \in S_P^R \text{ such that } l_k \le L - x_R, \\ W(L - x_R) & \text{if } \nexists k,\ k \in S_P^R \text{ such that } w_k \le W - y_R, \\ LW - x_R y_R & \text{if } \exists (k \wedge k') \in S_P^R \text{ such that} \\ & (w_k \le W - y_R) \wedge (l_{k'} \le L - x_R), \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

The tightest of $\mathcal{U}_3(x_R, y_R)$ and $\mathcal{U}_4(x_R, y_R)$, is a valid upper bound:

$$\mathcal{U}_5(x_R, y_R) = \min\{\mathcal{U}_3(x_R, y_R), \mathcal{U}_4(x_R, y_R)\}. \tag{7}$$

Comparing $\mathcal{U}_5(x_R, y_R)$ to $\mathcal{U}_1(x_R, y_R)$ and to an estimate based on $\mathcal{U}_2(x_R, y_R)$, gives rise to a tighter upper bound

$$\mathcal{U}_6(x_R, y_R) = \min\{\mathcal{U}_1(x_R, y_R), \mathcal{U}_2(L, W) - g(R), \mathcal{U}_5(x_R, y_R)\} \tag{8}$$

$$= \min\{\mathcal{U}_1(x_R, y_R), \mathcal{U}_2(L, W) - g(R), \mathcal{U}_3(x_R, y_R), \mathcal{U}_4(x_R, y_R)\}. \tag{9}$$

Computing $\mathcal{U}_6(.,.)$ requires the same computational effort needed to determine $\mathcal{U}_5(.,.)$. To speed the computation of $\mathcal{U}_5(.,.)$, we use when applying (6) a normalized rectangle; which reduces the size of T.

- For instance, the area P of Fig. 4(a) can be reduced to the normalized area $P1$ of Fig. 4(b), which is the union of a horizontal strip $(L - x_R, W)$ and a vertical strip $(L, W - w_R)$. The chosen horizontal strip is the widest feasible horizontal strip that can be constructed using the current set of pieces.

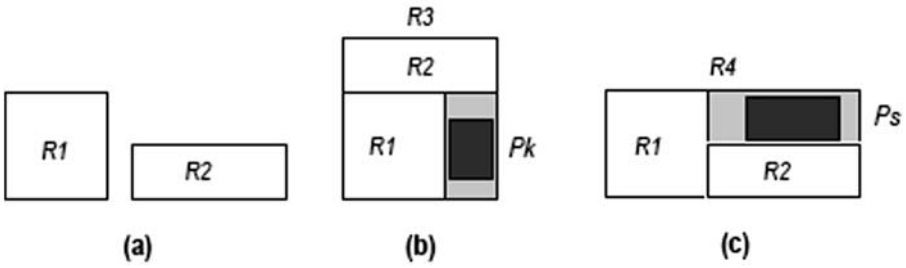

Fig. 5 Horizontal and vertical examples of dominated internal rectangles

- The normalized area $P1$ of Fig. 4(b) can be further reduced into a single normalized strip as shown in Fig. 4(c) where the vertical strip is empty.

4.3 Development strategies

The performance of a branch-and-bound procedure highly depends on its particular implementation. Employing sophisticated techniques can drastically accelerate a branch and bound implementation. To speed up our exact algorithm, we reduce the size of the search space by discarding (i) dominated, (ii) duplicate, and (iii) non feasible patterns, and by (iv) adopting an ordered data structure for *Open* and *Clist*.

4.3.1 Dominated patterns

Consider pattern $R1$ taken from the *Open* list and $R2$ stored in the *Clist*. $R3 = \frac{R1}{R2}$ is a dominated pattern if there exists a piece k such that $(l_k, w_k) \leq (l_{R3} - l_{R1}, w_{R1})$ (or $(l_k, w_k) \leq (l_{R3} - l_{R2}, w_{R2})$) and $b_k - d_k^{R3} \geq 1$, with d_k^{R3} being the number of occurrences of the piece type k in $R3$. Similarly, $R4 = R1|R2$, is a dominated pattern if there exists a piece s such that $(l_s, w_s) \leq (l_{R2}, w_{R4} - w_{R2})$ (or $(l_s, w_s) \leq (l_{R1}, w_{R4} - w_{R1})$) and $b_s - d_s^{R4} \geq 1$, with d_s^{R4} being the number of occurrences of the piece type s in $R4$. The patterns $R3$ and $R4$ of Figs. 5(b–c) are dominated patterns.

4.3.2 Duplicate patterns

The patterns of Fig. 6(b) are symmetric. They have the same size and use the exact same set of internal rectangles (or initial pieces); subsequently, these patterns have equal internal values. They are duplicate patterns. They can be easily detected as follows. Let A be an element taken from *Open* and B an element of *Clist*. The horizontal build $A|B$ is a duplicate internal rectangle if there exists another internal rectangle $C = (l_C, w_C)$ in *Open* or *Clist* such that

1. $(l_C, w_C) \leq (l_A + l_B, \max\{w_A, w_B\})$, and
2. $\forall i \in \{1, \ldots, n\}$, $d_i^{A|B} \leq d_i^C$, where $d_i^{(\bullet)}$ denotes the number of occurrences of piece type i in the current internal rectangle.

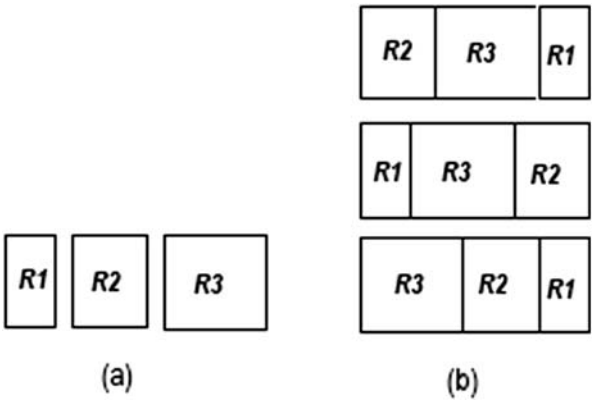

Fig. 6 **a** Three internal rectangles R_1, R_2 and R_3; each representing a piece or an internal rectangle, and **b** three duplicate patterns each obtained by a horizontal build of R_1, R_2 and R_3

4.3.3 Non feasible patterns

At each node of the developed tree, corresponding to a new internal rectangle R of dimensions $x_R \times y_R$, a set of upper bounds is computed. These bounds curtail the search space by fathoming nodes that yield infeasible solutions. Among these bounds is the upper bound $\mathcal{U}_4(x_R, y_R)$ which is obtained by solving $SKBK_{(x_R,y_R)}$.

The following two propositions are used to discard infeasible nodes when a horizontal/vertical build of an element of *Open* and an element of *Clist* is considered.

Proposition 4.1 *The horizontal* (*resp. vertical*) *build of a pattern A taken from Open and a pattern B taken from Clist is discarded if*

$$\sum_{k \in P_R^P}^{n} l_k w_k a'_k > T,$$

where $a'_k = a_k - d_k^R$, *and* T *is the capacity of* $SKBK_{(x_R,y_R)}$ *for internal rectangle* R.

Proof The horizontal build of A and B produces an internal rectangle R of value $g(R)$ and an occupied area equal to $\sum_{k \in R} l_k w_k d_k^R$.

Let S' be the set of pieces whose $a_i - d_i^R > 0$, and for which T satisfies (6). The smallest area used by the pieces of S' is equal to $T' = \sum_{i \in S'} l_i w_i (a_i - d_i^R)$. If $T' \leq T$, each piece type $i \in S'$ satisfies its lower demand constraint. On the other hand, if $T' > T$, then there exists a subset S'' such that

$$T' = T + \sum_{k \in S''} l_k w_k (a_k - d_k^R).$$

That is, there are piece types $k \in S''$ whose lower demand constraints are not satisfied. Hence, the problem does not admit a feasible solution when R is constructed. □

Proposition 4.2 *Let R be the build of two patterns A and B. R does not produce a feasible solution to the DCTDC problem if*

$$\sum_{k\in I} l_k w_k a'_k > T.$$

Proof $d_k^R = 0$ if piece type k does not satisfy (6). Therefore, we distinguish two cases: (i) the capacity T is saturated by *BKSP* and (ii) T is not saturated.

Case (i) implies that the set S' representing the pieces entering S_P^R satisfies the lower demand constraints for those pieces. If there exists a piece $t \notin S'$ such that $a_t - d_t^R > 0$, then its lower demand constraint is violated, and the solution is infeasible.

Case (ii) implies that all pieces of S' satisfy their respective lower demand constraints. Therefore, the solution constructed by positioning R at the bottom-left corner is not feasible if problem $\overline{RSBK}_{(x_R,y_R)}$ does not saturate the capacity $\overline{T} = T - \sum_{k\in S'} l_k w_k (a_k - d_k^R)$, where

$$(\overline{RSBK}_{(x_R,y_R)}) \begin{cases} \max & \displaystyle\sum_{i\in S''} l_i w_i z_i, \\ \text{subject to} & \displaystyle\sum_{i\in S''} (l_i w_i) z_i \le \overline{T},\ z_i \le a_i,\ z_i \ge 0,\ i \in S'', \end{cases}$$

and S'' denotes the set of pieces that do not satisfy (6) and whose $d_i^R = 0$.

Since there exists at least a piece $t \in S''$ for which a_t is violated, then the upper bound $g(R) + \mathcal{U}_4(x_R, y_R)$ is the value of a non-feasible solution to the DCTDC problem. □

4.3.4 Data structure

To improve the performance of the algorithm, we use a particular *data structure* for *Clist* which stores *candidates* to the optimal solution. We sort all cutting patterns (x_R, y_R) in a non-decreasing order of both their lengths and widths. The range of possible lengths and widths is limited by (L, W), the dimensions of the stock rectangle S. The ordered data structure facilitates the update of the temporary list Q, containing all horizontally/vertically built elements. If R is the current element selected from *Open* and moved to *Clist*, and Q_{l_R} and Q_{w_R} are the two sets of Clist containing candidate elements for combinations with R such that

$$Q_{l_R} = \{q \mid l_q = l_R + l_p \le L,\ p \in \text{Clist}\}$$

and

$$Q_{w_R} = \{q \mid w_q = w_R + w_p \le W,\ p \in \text{Clist}\},$$

then, $Q = Q_{l_R} \cup Q_{w_R}$.

4.4 Remarks

There might be cases when IGP does not yield a feasible solution to the DCTDC problem. In this case, two alternative approaches can be undertaken. The first alternative starts the exact algorithm with an initial (non)feasible solution whose value is equal to $LB = (\sum_{i=1}^{n} a_i) - 1$. If the algorithm exits with the same initial solution, then the problem has no feasible solution; else, the incumbent solution is optimal. The second alternative starts the exact algorithm using depth first. Once a "good" initial feasible solution is obtained, then the exact algorithm is restarted with that solution as a lower bound.

5 Computational results

The purpose of this section is twofold. First, we assess the quality of the approximate solutions to the DCTDC obtained with GP and IGP. Second, we evaluate the performance of the exact algorithm ALGO. All approaches were coded in C++ and run on a multi-users Sun UltraSparc1 (450 MHz, 512 Mb), with a thirty minute CPU time limit. The approaches were tested on two sets of instances.

5.1 Results for the first set of problems

The first set of instances[1] has: (i) 14 unweighted instances ($\forall i \in I$, $c_i = l_i \times w_i$), and (ii) 10 weighted instances ($\exists i \in I$, $c_i \neq l_i \times w_i$). These instances were generated as follows. The dimensions L and W of the initial stock rectangle are randomly generated from the integer uniform [40, 150]. The integer number n of pieces to cut is randomly selected from the discrete uniform [10, 30], and the dimensions l_i and w_i follow the uniform $[0.1L, 0.85L]$, and $[0.1W, 0.85W]$, respectively. The weight c_i associated to a piece type i is $c_i = \lceil \alpha l_i w_i \rceil$, where $\alpha = 1$ for unweighted instances and α follows a uniform [0.25, 0.75] for weighted instances. The upper (resp. lower) demand b_i (resp. a_i), for $i = 1, \ldots, n$, have been chosen such that $b_i = \min\{\alpha_1, \alpha_2\}$, where $\alpha_1 = \lfloor \frac{L}{l_i} \rfloor \lfloor \frac{W}{w_i} \rfloor$, α_2 is a randomly generated integer in the interval $[a_i, 10]$, and a_i a randomly generated integer from the discrete Uniform [0, 4].

5.1.1 Performance of GP and IGP

Herein, we evaluate the performance of GP and IGP by comparing their respective solutions LB1 and LB2 to the exact solution Opt provided by ALGO.

For each instance, Table 1 shows Opt, LB1, LB2, the experimental approximation ratios $R_{\mathrm{LB}i} = \frac{\mathrm{LB}i}{\mathrm{Opt}}$, $i = 1, 2$, the number of times the procedure that restores feasibility was applied Res, and IGP's computational time T_2 measured in seconds. Table 1 does not report the runtime of GP, which is less than 0.01. Similarly, it does not report the runtime of the procedure used to restore feasibility since it is of order of zero.

[1] Available from `http://www.laria.u-picardie.fr/hifi/OR-Benchmarks`.

Table 1 Quality of the lower bounds provided by heuristics GP and IGP. The symbol h (resp. v) means that the lower bound is obtained using horizontal (resp. vertical) strips. The couple (h, v) means that the final solution is obtained by applying a horizontal construction completed with a vertical one. The value a (resp. b) represents $\sum_{i=1}^{n} a_i$ (resp. $\sum_{i=1}^{n} b_i$)

Instance				First lower bound		Second lower bound			
#Inst	a	b	Opt	LB1	R_{LB1}	LB2	R_{LB2}	Res	T_2
du1	7	22	14317	12934^{h}	0.90	$13490^{(h,v)}$	0.94	204	1.21
du2	3	34	14735	12404^{h}	0.84	$14145^{(h,v)}$	0.96	126	1.31
du3	6	35	14646	12563^{h}	0.86	$14010^{(h,v)}$	0.96	105	2.11
du4	4	45	7078	6664^{h}	0.94	$6932^{(h,v)}$	0.98	65	1.09
du5	5	40	14114	12047^{h}	0.85	$13478^{(h,v)}$	0.95	176	2.02
du6	10	46	15278	14647^{h}	0.96	$14647^{(h,v)}$	0.96	143	1.32
du7	12	46	15473	12146^{h}	0.78	$14064^{(h,v)}$	0.91	232	1.29
du8	13	49	5301	4404^{h}	0.83	$5301^{(h,v)}$	1.00	243	0.66
du9	7	48	12715	11474^{h}	0.90	$12340^{(h,h)}$	0.97	187	1.41
du10	7	49	14947	14016^{h}	0.94	$14602^{(h,v)}$	0.98	194	1.23
du11	6	71	15504	12454^{h}	0.80	$14785^{(h,v)}$	0.95	276	2.08
du12	12	71	18037	15029^{v}	0.83	$17049^{(v,h)}$	0.95	67	2.21
du13	6	65	20427	13801^{h}	0.68	$17353^{(h,v)}$	0.85	234	2.01
du14	11	69	16762	13596^{h}	0.81	$15058^{(h,v)}$	0.90	198	1.75
dw1	7	33	10296	9020^{h}	0.88	$9925^{(h,v)}$	0.96	201	1.1
dw2	5	34	10062	9219^{v}	0.92	$9442^{(v,v)}$	0.94	109	2.02
dw3	7	34	10312	9501^{v}	0.92	$9914^{(v,h)}$	0.96	232	1.54
dw4	4	33	8478	8314^{h}	0.98	$8384^{(v,h)}$	0.99	120	2.03
dw5	6	45	3660	3378^{v}	0.92	$3385^{(h,v)}$	0.92	43	2.01
dw6	9	55	8099	7524^{v}	0.93	$8028^{(h,v)}$	0.99	123	2.26
dw7	8	62	7171	6877^{v}	0.96	$7081^{(h,v)}$	0.99	143	2.15
dw8	7	52	9655	8675^{h}	0.90	$8739^{(h,v)}$	0.91	98	2.41
dw9	6	111	4550	4450^{h}	0.98	$4550^{(h,h)}$	1.00	105	2.34
dw10	8	111	8699	8360^{h}	0.96	$8414^{(h,v)}$	0.97	22	2.53
Av. of Unweighted					0.85		0.95		1.55
Av. of Weighted					0.93		0.96		2.04
Average					0.89		0.95		1.75

The quality of the solution reached by GP is in few cases far from optimal. Column 3 of Table 1 shows that the average approximation ratio is 0.85 (resp. 0.93) for unweighted (resp. weighted) instances, and 0.89 for all instances. The computational time of GP is very short; it is less than 0.01 seconds for all instances. The short runtime of GP and its occasional unsatisfactory performance are inciting factors to couple it with a discretization procedure such as IGP. The average number of times the procedure that restores feasibility is applied depends, in general, on the linear combinations of widths used by GP. The procedure was applied 168.58 times on average over all instances.

IGP improves the quality of the solutions yielded by GP as illustrated by Columns 7 and 8. The comparison of Columns 6 and 8 reveals the increase of the

Table 2 Impact of the implementation strategies on the exact approach. The symbol ∘ means that the algorithm does not reach the optimal solution after 30 minutes of runtime

#Inst	Opt	Algo1		Algo2		Algo3		Algo4	
		Node	Time	Node	Time	Node	Time	Node	Time
du1	14317	199878	∘	185961	∘	159941	1304.17	16583	20.03
du2	14735	66843	∘	341241	∘	233401	∘	73153	420.66
du3	14646	76853	567.28	76853	567.28	76476	451.98	12474	7.14
du4	7078	342087	∘	231661	∘	101282	1204.64	66811	324.31
du5	14114	345085	∘	343191	∘	343088	∘	89125	1234.51
du6	15278	47710	157.45	29953	148.56	29953	148.56	6766	1.52
du7	15473	206809	473.72	206753	471.72	191386	468.55	36863	103.91
du8	5301	70292	83.72	67292	56.72	62192	47.72	12160	9.19
du9	12715	340219	∘	345256	∘	105439	1345.32	40350	124.48
du10	14947	341857	∘	343887	∘	343522	∘	43573	157.24
du11	15504	145141	∘	145121	1420.18	99580	98.12	28247	53.09
du12	18037	131642	∘	131642	1675.73	118529	104.16	26075	43.45
du13	20427	230205	∘	220205	1100.34	219322	556.5	52253	192.41
du14	16762	271554	920.24	253718	801.98	267262	326.08	50045	173.45
dw1	10296	185808	733.14	182957	643.95	175735	545.36	30339	79.08
dw2	10062	6677	13.31	4626	8.67	2540	3.67	456	0.14
dw3	10312	345790	∘	330864	∘	230547	1022.28	24154	44.98
dw4	8478	88624	295.38	48050	183.44	29618	84.56	6481	2.12
dw5	3660	161704	∘	121070	1127.06	119577	1022.91	32008	149.72
dw6	8099	78402	275.28	75246	274.5	50728	165.69	5921	2.61
dw7	7171	110448	∘	187198	∘	189056	∘	28168	68.09
dw8	9655	53027	1237.14	51564	241	50484	238.31	5186	2.25
dw9	4550	349237	∘	78095	395.26	23165	135.94	8593	13.56
dw10	8699	217587	445.58	109958	265.72	87579	247.3	17284	20.95
Av. Unweighted		118485.73	472.93	100633.64	333.05	93086.64	247.98	16725.00	36.58
Av. Weighted		105020.83	499.97	78733.50	269.55	66114.00	214.15	10944.50	17.86
Average		118485.73	472.93	100633.64	333.05	93086.64	247.98	16725.00	36.58

global approximation ratio from 0.89 to 0.95. This sizable improvement costs less than 2 seconds of additional computational time.

The performance of both GP and IGP is better for weighted instances than for unweighted instances; though their runtimes are relatively longer for weighted instances than for unweighted ones.

5.1.2 Performance of the exact algorithm

The performance of the exact algorithm, in terms of number of solved instances and run time, depends on the quality of the initial starting solution and the adopted search strategy. We consider four versions of the exact algorithm.

1. The first version, ALGO1, uses $\mathcal{U}_1(x_R, y_R)$, and $\mathcal{U}_2(x_R, y_R)$.
2. The second version, ALGO2, uses $\mathcal{U}_6(x_R, y_R)$, the enhanced upper bound.
3. The third version, ALGO3, applies ALGO2 with detection of both dominated and duplicate patterns.
4. The fourth and last version, ALGO4, extends ALGO3 by including the non-feasibility pruning.

ALGO1, as illustrated by Table 2, solves 45.83% of the instances. Its average computational time on solved instances is close to 473 seconds. Its average number of nodes is the largest among the four versions. ALGO2 improves ALGO1 solving more instances; in fact, it solves 66.67% of the instances. Its average computational time on solved instances, which is 333 seconds, is shorter than the average computational time of ALGO1. This is mainly due to its reduced average number of nodes. Fathoming dominated and duplicate patterns further accelerates the search process and enhances the performance of the algorithm. Indeed, ALGO3 solves almost three times as many instances as ALGO1 and almost twice as many instances as ALGO2. It solves 83.33% of the instances with a reduced runtime that equals on average 248 seconds. Injecting the non-feasibility pruning induces a better version of the algorithm which obtains the optimal solution for all instances with a reduced runtime that equals on average 36 seconds.

To study the impact of the starting solution on the search tree of ALGO4 and its runtime, we run ALGO4 (i) without any lower bound, (ii) with LB1, and (iii) with LB2 as the starting solution of the branch and bound search tree. Table 3 shows that ALGO4 with LB1 as the initial solution is on average faster than ALGO4 without a lower bound and solves more instances (only four unsolved instances versus eight unsolved instances). Using LB2 as the initial solution further enhances the algorithm. Not only does it solve all instances, but it also decreases the computational time of solved instances from 226.9 to 48.71 seconds, and the computational time of all instances from 570.37 to 135.37 seconds.

5.2 Results for the second set of problems

Furthermore, we evaluated the performance of the proposed exact algorithm on 40 additional instances from the literature. Most of these instances, which tackle various sized problems, were designed for the non-guillotine (or orthogonal) two-dimensional problem. Specifically, they are composed of twelve instances of *gcut* type (gcut1, ..., gcut12), six instances of *hadchr* type (taken from [8]), seven instances of *ngcutcon* type (ngcutcon11, ..., ngcutcon17), ten instances of *ngcutfs* type, and five instances of *okp* type (taken from [18]). Evidently, since these instances did not tackle the DCTDC problem, their optimal solution to the current problem is unknown. Therefore, we first run all these instances with $a_i = 0$, $i \in I$, and compare the obtained optimal solution OptCG to that of the general problem OptCNG (without the guillotine cuts requirement). Whenever OptCNG is not available, we solve a knapsack in order to obtain an upper bound for the problem. The comparison of OptCG and OptCNG reveals how tight the guillotine constraints are for each instance. For instances where the length of the initial stock plate is larger than 500, the upper bound

Table 3 Behavior analysis of the fourth version of the algorithm when varying the starting solution

#Inst	Opt	Algo4-Without LB		Algo4-With LB1		Algo4-With LB2	
		Node	Time	Node	Time	Node	Time
du1	14317	49457	376.75	25471	145.56	16583	20.03
du2	14735	10017	∘	10009	∘	73153	420.66
du3	14646	43876	165.43	12599	54.57	12474	7.14
du4	7078	117412	∘	81939	∘	66811	324.31
du5	14114	160032	∘	161957	∘	89125	1234.51
du6	15278	11028	19.47	7864	8.76	6766	1.52
du7	15473	40120	492.20	40110	463.32	36863	103.91
du8	5301	16160	44.76	16160	40.76	12160	9.19
du9	12715	96520	∘	65109	1202.76	40350	124.48
du10	14947	130883	∘	101481	∘	43573	157.24
du11	15504	46210	404.51	36209	346.08	28247	53.09
du12	18037	38876	276.78	28280	202.00	26075	43.45
du13	20427	75432	987.61	52289	775.80	52253	192.41
du14	16762	90063	921.54	50738	721.12	50045	173.45
dw1	10296	62348	427.86	41791	388.65	30339	79.08
dw2	10062	15808	24.95	7534	9.32	456	0.14
dw3	10312	89785	∘	44914	654.64	24154	44.98
dw4	8478	16523	66.80	9090	12.68	6481	2.12
dw5	3660	103505	∘	42368	706.76	32008	149.72
dw6	8099	11215	22.64	10631	20.36	5921	2.61
dw7	7171	40278	642.92	29315	294.32	28168	68.09
dw8	9655	15284	19.65	8223	12.34	5186	2.25
dw9	4550	183978	∘	25243	294.32	8593	13.56
dw10	8699	37789	453.20	21121	134.76	17284	20.95
Av. of Unweighted		45691.33	409.89	29968.89	306.44	26829.56	67.13
Av. of Weighted		28463.57	236.86	18243.57	124.63	13405.00	25.03
Average		38154.19	334.19	24839.06	226.90	20956.31	48.71

based on Gilmore and Gomory's procedure can not be computed; thus, the algorithm uses only the upper bound obtained by solving the largest knapsack problem.

Second, we rerun our exact algorithm on each instance but with a_i being randomly generated and $a_i \leq b_i$, $i \in I$. Our comparison of the obtained solution OptCGD to OptCG shows the effect of lower demand constraints on the optimal solution values.

Third and last, we evaluate the effect of the initial lower bounds (which were not tight for the first set of instances and time-consuming in some instances) on the performance of the exact algorithm both in terms of runtime and number of nodes. Specifically, we evaluate the quality of LB2, which is the solution obtained by IGP, by comparing it to OptCGD and assessing the run time of IGP. We finally run the exact algorithm ALGO4 with and without the initial solution obtained by IGP, and compare its performance in terms of run time and number of nodes.

Columns 2–4 of Table 4 display OptCNG, OptCG and the percent deviation of OptCG from OptCNG. In 14 out of 40 instances, OptCG = OptCNG which indicates in these cases that one of the optimal solutions to the general case is guillotine. However, this is not the case for all instances, since the maximum deviation reached 17.91% for ngcutcon14 and 12.55% for ngcutcon13, whereas the average deviation equals 2.45%. As expected, reinforcing the guillotine constraint reduces the value of the optimum solution. Indeed, OptCG $\leq$ OptCNG for all instances.

Column 5 of Table 4 displays OptCGD obtained when introducing lower demand constraints while Column 6 tallies the percent deviation of the obtained solution from OptCG. As expected, OptCG is an upper bound for OptCGD with the average percent deviation equal to 6.17%. This percent deviation reaches 25.90% for hadchr7. There are though instances with a zero percent deviation; in these cases (hadchr2, ngcutcon13, and okp4), the optimal solution corresponding to OptCG satisfies the lower demand constraints.

Column 7 of Table 4 displays LB2, which is the value of the solution given by IGP, and used by our exact approach as an initial lower bound. Column 8 displays the number of times the procedure restoring feasibility is applied. The comparison of LB2 to OptCGD shows that the average gap equals 2.97%, but can be as large as 8.59% for instance gcut5 and 7.2% for gcut7. (There are of course cases when this gap is null.) The gap is in most cases very low; which further justifies the use of the lower bound especially that the run times are small. The analysis of Column 9 of Table 4 shows that the average runtime of IGP equals 1.36 seconds with a maximum observed run time equal to 4.56 seconds.

Columns 10–11 (resp. 12–13) display the number of nodes and the run time of ALGO4 without (resp. with) the initial solution obtained by IGP as a lower bound. ALGO4 did not reach the optimal solution within the 30 minutes allocated runtime in 6 out of the 40 instances when LB2 was not used as a lower bound whereas it converged to the optimal solution in all 40 instances when the lower bound was used. In addition, the number of generated nodes was drastically reduced with the reduction averaging 33.31%. That is, despite its simplicity, the lower bound fathoms a sizable part of the search tree; speeding the search and allowing ALGO4 to solve all instances. Indeed, not accounting for the cases that could not be solved within the 30-minute threshold, the average run time of ALGO4 was reduced from 115.99 seconds to 56.39 seconds with the maximum run time being reduced from 1786 seconds to 895 seconds. Accounting for the unsolved cases will obviously further emphasize the time reduction. It can therefore be concluded that the quality of the starting solution does influence the performance of the algorithm. Subsequently, investigating procedures yielding tighter lower bounds seems important in this context.

6 Conclusion

In this paper, we studied the double constrained two dimensional layout problem where the number of times each piece type appears is bounded by a lower and an upper demand. We solved the problem using an exact branch and bound procedure with efficient bounding schemes. An initial lower bound is obtained using a two-stage

Table 4 A comparative study between guillotine and non-guillotine quality solutions and behavior analysis of the last version of the algorithm with and without a starting solution

#Inst	%Dev			%Dev		LB2	*Res*	*Time*	Algorithm ALGO4			
	OptCNG	OptCG	NG vs G	OptCGD	G vs GD				Without LB2		With LB2	
									Node	*Time*	*Node*	*Time*
gcut1	48368	48368	0.00	40625	16.01	40625	44	0.45	50	0.80	22	0.68
gcut2	59798	59307	0.82	57996	2.21	54450	53	0.86	558	2.01	172	1.82
gcut3	61275	60241	1.69	58677	2.60	58438	128	1.23	1784	3.91	485	2.94
gcut4	61380	60942	0.71	58866	3.41	58004	150	2.54	7509	28.35	1973	9.47
gcut5	195582	195582	0.00	183339	6.26	167594	218	1.43	435	3.60	89	2.86
gcut6	236305	236305	0.00	189391	19.85	178443	354	1.65	377	7.20	302	6.80
gcut7	240143	238974	0.49	217082	9.16	201445	443	2.76	741	10.64	483	9.70
gcut8	245758	245758	0.00	239195	2.67	232764	362	1.04	7292	37.40	2254	19.97
gcut9	939600	919476	2.14	874589	4.88	858697	492	2.02	124	13.90	77	14.20
gcut10	937349	903435	3.62	790985	12.45	774901	275	3.05	787	24.02	220	21.70
gcut11	969709	955389	1.48	919069	3.80	890177	301	3.32	2247	37.80	973	32.50
gcut12	979521	970744	0.90	869700	10.41	861170	295	3.67	7298	46.98	2564	36.72
hadchr2	430	430	0.00	430	0.00	430	19	0.01	34	0.01	5	0.01
hadchr3	1178	1178	0.00	963	18.25	914	18	0.01	77	0.05	21	0.02
hadchr5	289	289	0.00	275	4.84	256	29	0.01	425	0.08	307	0.05
hadchr6	924	924	0.00	732	20.78	720	34	0.01	1099	0.53	1049	0.49
hadchr7	1865	1865	0.00	1382	25.90	1382	56	0.01	1233	0.41	1142	0.39
hadchr11	1270	1270	0.00	1188	6.46	1147	27	0.02	863	0.46	751	0.37
ngcutcon11	1864	1688	9.44	1518	10.07	1518	41	0.01	530	0.11	227	0.08
ngcutcon12	2012	1865	7.31	1672	10.35	1672	65	0.05	13331	20.81	3456	7.85
ngcutcon13	1347	1178	12.55	1178	0.00	1178	17	0.01	94	0.09	26	0.05

Table 4 (*continued*)

#Inst	%Dev			%Dev		LB2	*Res*	*Time*	Algorithm ALGO4			
	OptCNG	OptCG	NG vs G	OptCGD	G vs GD				Without LB2		With LB2	
									Node	*Time*	*Node*	*Time*
ngcutcon14	1547	1270	17.91	1216	4.25	1200	28	0.01	742	0.26	489	0.20
ngcutcon15	2800	2721	2.82	2700	0.77	2614	6	0.01	429	0.15	89	0.06
ngcutcon16	2020	1860	7.92	1700	8.60	1620	34	0.53	5432	13.70	2179	5.10
ngcutcon17	29133	27589	5.30	25377	8.02	23609	81	3.24	84264	∘	43836	823.52
ngcutfs1.1	29955	27968	6.63	26542	5.10	25204	123	0.85	12355	85.70	7053	34.60
ngcutfs1.2	30000	29181	2.73	28807	1.28	27377	6	1.04	9500	42.80	6965	36.80
ngcutfs1.3	30000	29865	0.45	29830	0.12	28754	110	1.65	7845	7.70	3601	4.90
ngcutfs1.4	30000	29964	0.12	29628	1.12	29361	15	2.41	98181	∘	29864	956.70
ngcutfs1.5	30000	29961	0.13	29891	0.23	29454	104	3.65	107235	1785.98	76984	895.06
ngcutfs2.1	29973	27659	7.72	26731	3.36	25324	98	1.11	86364	∘	59034	1094.56
ngcutfs2.2	30000	28984	3.39	28327	2.27	28093	123	1.21	61066	∘	33379	807.15
ngcutfs2.3	30000	29847	0.51	29325	1.75	29100	176	1.89	28206	336.25	7579	85.74
ngcutfs2.4	30000	29833	0.56	29122	2.38	27595	282	3.01	98098	∘	46366	1320.35
ngcutfs2.5	30000	29970	0.10	29535	1.45	29408	0	4.56	74984	∘	51195	1333.68
okp1	27718	27589	0.47	26372	4.41	25963	0	0.09	11775	34.04	5731	16.90
okp2	22502	22502	0.00	21916	2.60	20641	0	1.31	19188	247.64	11776	144.22
okp3	24019	24019	0.00	23740	1.16	23655	75	0.85	12354	82.65	8984	61.52
okp4	32893	32893	0.00	32893	0.00	30554	124	1.13	123453	987.83	76876	443.76
okp5	27923	27923	0.00	25835	7.48	25298	107	1.54	14371	79.83	6557	19.70
Average			2.45		6.17			1.36	22568.25	115.99	12378.38	56.39

procedure while upper bounds are computed using a variety of knapsack problems. The proposed approach solves small and medium sized instances in a very short computational time. Its performance is mainly due to the particular implementation of the branch and bound algorithm. Indeed, the adopted data structure and symmetry, duplicate, and non-feasibility detection strategies, which fathom unnecessary branches, drastically reduce the size of the search tree; subsequently, enhancing the performance of the algorithm.

References

1. Baker, B.S., Coffman, E.G. Jr., Rivest, R.L.: Orthogonal packing in two dimensions. SIAM J. Comput. **9**, 846–855 (1980)
2. Beasley, J.E.: Algorithms for unconstrained two-dimensional guillotine cutting. J. Oper. Res. Soc. **36**, 297–306 (1985)
3. Belov, G., Scheithauer, G.: A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting. Eur. J. Oper. Res. **171**, 85–106 (2006)
4. Blazewicz, J., Moret-Salvador, A., Walkowiak, R.: Parallel tabu search approaches for two-dimensional cutting. Parallel Process. Lett. 14, 23–32
5. Bortfeldt, A.: A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces. Eur. J. Oper. Res. (2005), available online
6. Caprara, A., Monaci, M.: On the 2-dimensional knapsack problems. Oper. Res. Lett. **32**, 5–14 (2004)
7. Christofides, N., Whitlock, C.: An algorithm for two-dimensional cutting problems. Oper. Res. **25**, 31–44 (1977)
8. Christofides, N., Hadjiconstantinou, E.: An exact algorithm for orthogonal 2-D cutting problems using guillotine cuts. Eur. J. Oper. Res. **83**, 21–38 (1995)
9. Cui, Y.: Generating optimal T-shape cutting patterns for rectangular blanks. Proc. Inst. Mech. Eng. Part B: J. Eng. Manuf. **218/B8**, 857–866 (2004)
10. Cui, Y., Wang, Z., Li, J.: Exact and heuristic algorithms for staged cutting problems. Proc. Inst. Mech. Eng. Part B: J. Eng. Manuf. **219/B2**, 201–208 (2005)
11. Cung, V.-D., Hifi, M.: Handling lower bound constraints in two-dimensional cutting problems. In: ISMP 2000, The 17th Symposium on Mathematical Programming, Atlanta, 7–11 August 2000
12. Cung, V.-D., Hifi, M., Le Cun, B.: Constrained two-dimensional cutting stock problems: a best-first branch-and-bound algorithm. Int. Trans. Oper. Res. **7**, 185–210 (2000)
13. Cung, V.-D., Hifi, M., Le Cun, B.: Constrained two-dimensional cutting stock problems: the NMVB approach and the duplicate test revisited. Working Paper, Série Bleue No 2000.127 (CERMSEM), Maison des Sciences Economiques, Université Paris 1 (2000)
14. Dyckhoff, H.: A typology of cutting and packing problems. Eur. J. Oper. Res. **44**, 145–159 (1990)
15. Dikili, A.C.: A new approach for the solution of the two-dimensional guillotine-cutting problem in ship production. Ocean Eng. **31**, 1193–1203 (2004)
16. Fayard, D., Zissimopoulos, V.: An approximation algorithm for solving unconstrained two-dimensional knapsack problems. Eur. J. Oper. Res. **84**, 618–632 (1995)
17. Fayard, D., Hifi, M., Zissimopoulos, V.: An efficient approach for large-scale two-dimensional guillotine cutting stock problems. J. Oper. Res. Soc. **49**, 1270–1277 (1998)
18. Fekete, S.P., Schepers, J.: A general framework for bounds for higher-dimensional orthogonal packing problems. Math. Method. Oper. Res. **60**, 311–329 (2004)
19. Gilmore, P., Gomory, R.: Multistage cutting problems of two and more dimensions. Oper. Res. **13**, 94–119 (1965)
20. Gilmore, P., Gomory, R.: The theory and computation of knapsack functions. Oper. Res. **14**, 1045–1074 (1966)
21. Herz, J.C.: A recursive computing procedure for two-dimensional stock cutting. IBM J. Res. Dev. **16**, 462–469 (1972)
22. Hifi, M.: An improvement of Viswanathan and Bagchi's exact algorithm for cutting stock problems. Comput. Oper. Res. **24**, 727–736 (1997)
23. Hifi, M., M'Hallah, R.: Strip generation algorithms for two-staged two-dimensional cutting stock problems. Eur. J. Oper. Res. **172**, 515–527 (2006)

24. Hifi, M., M'Hallah, R.: An exact algorithm for constrained two-dimensional two-staged cutting problems. Oper. Res. **53**, 140–150 (2005)
25. Hifi, M., Zissimopoulos, V.: A recursive exact algorithm for weighted two-dimensional cutting. Eur. J. Oper. Res. **91**, 553–564 (1996)
26. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer, Berlin (2004). ISBN:3-540-40286-1
27. Lodi, A., Martello, S., Monaci, M.: Two-dimensional packing problems: A survey. Eur. J. Oper. Res. **141**, 241–252 (2002)
28. Lodi, A., Monaci, M.: Integer linear programming models for 2-staged two-dimensional Knapsack problems. Math. Program. **94**, 257–278 (2003)
29. Morabito, R., Arenales, M.: Staged and constrained two-dimensional guillotine cutting problems: An and/or-graph approach. Eur. J. Oper. Res. **94**, 548–560 (1996)
30. Mumford-Valenzuela, C.L., Vick, J., Wang, P.Y.: Heuristics for large strip packing problems with guillotine patterns: An empirical study. In: Metaheuristics: Computer Decision-Making, pp. 501–522. Kluwer Academic, Dordrecht (2003)
31. Suliman, S.M.A.: A sequential heuristic procedure for the two-dimensional cutting-stock problem. Int. J. Prod. Econ. **99**, 177–185 (2006)
32. Viswanathan, K.V., Bagchi, A.: Best-first search methods for constrained two-dimensional cutting stock problems. Oper. Res. **41**, 768–776 (1993)
33. Wang, P.Y.: Two algorithms for constrained two-dimensional cutting stock problems. Oper. Res. **31**, 573–586 (1983)
34. Wäescher, G., Haussner, H., Schumann, H.: An improved typology of cutting and packing problems. Eur. J. Oper. Res. **183**, 1109–1130 (2007)