CIRRELT
Centre interuniversitaire de recherche
sur les réseaux d'entreprise, la logistique et le transport
Interuniversity Research Centre
on Enterprise Networks, Logistics and Transportation

Combinatorial Benders' Cuts for the Strip Packing Problem

Jean-François Côté
Mauro Dell'Amico
Manuel Iori

April 2013

CIRRELT-2013-27

Bureaux de Montréal :
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal (Québec)
Canada H3C 3J7
Téléphone : 514 343-7575
Télécopie : 514 343-7121

Bureaux de Québec :
Université Laval
2325, de la Terrasse, bureau 2642
Québec (Québec)
Canada G1V 0A6
Téléphone : 418 656-2073
Télécopie : 418 656-2624

www.cirrelt.ca

Combinatorial Benders' Cuts for the Strip Packing Problem

Jean-François Côté[1,*], Mauro Dell'Amico[2], Manuel Iori[2]

[1] Interuniversity Research Centre on Enterprise Networks, Logistics and Transportation (CIRRELT) and Department of Computer Science and Operations Research, Université de Montréal, P.O. Box 6128, Station Centre-Ville, Montréal, Canada H3C 3J7

[2] DISMI, University of Modena and Reggio Emilia, Via Amendola 2, Padiglione Morselli, 42122, Reggio Emilia, Italy

Abstract. We study the strip packing problem, in which a set of two-dimensional rectangular items has to be packed in a rectangular strip of fixed width and infinite height, with the aim of minimizing the height used. The problem is important because it models a large number of real-world applications, including cutting operations where stocks of materials such as paper or wood come in large rolls and have to be cut with minimum waste, scheduling problems in which tasks require a contiguous subset of identical resources, and container loading problems arising in the transportation of items that cannot be stacked one over the other. The strip packing problem has been attacked in the literature with several heuristic and exact algorithms, but, nevertheless, benchmark instances of small size remain unsolved to proven optimality since many years. In this paper we propose a new exact method that solves a large number of the open benchmark instances within a limited computational effort. Our method is based on a Benders' decomposition, in which in the master we cut items into unit-width slices and pack them contiguously in the strip, and in the slave we attempt to reconstruct the rectangular items by fixing the vertical positions of their unit-width slices. If the slave proves that the reconstruction of the items is not possible, then a cut is added to the master, and the algorithm is re-iterated. We show that both the master and the slave are strongly *NP*-hard problems, and solve them with tailored pre-processing, lower and upper bounding techniques, and exact algorithms. We also propose several new techniques to improve the standard Benders' cuts, using the so-called combinatorial Benders' cuts, and an additional lifting procedure. Extensive computational tests show that the proposed algorithm provides a substantial breakthrough with respect to previously published algorithms.

Keywords: Strip packing problem, exact algorithm, Benders' decomposition, combinatorial Benders' cut.

* Corresponding author: Jean-Francois.Cote@cirrelt.ca

Dépôt légal – Bibliothèque et Archives nationales du Québec
Bibliothèque et Archives Canada, 2013

1 Introduction

In the *Strip Packing Problem* (SPP) we are given a set $\mathcal{N} = \{1, 2, \ldots, n\}$ of rectangular items of width w_j and height h_j, and a rectangular strip of width W and infinite height. The aim is to pack the items in the strip by minimizing the height used for the packing. Items cannot overlap, must be packed with their edges parallel to the borders of the strip, and cannot be rotated. A SPP solution is depicted in Figure 1-(a), where a set of seven items is packed in a strip of width $W = 10$, by using minimum height $z = 9$.

The SPP is important because it models a large number of real-world applications. It models cutting applications in the manufacturing industry, where stock of materials such as paper, wood, glass and metal come in large rolls and have to be cut by minimizing waste, see, e.g., Gilmore and Gomory (1965). It also models scheduling problems in which tasks require a contiguous subset of identical resources, see, e.g., Augustine et al. (2009), and packing problems arising in the transportation of items that cannot be stacked one over the other, see, e.g., Iori et al. (2007).

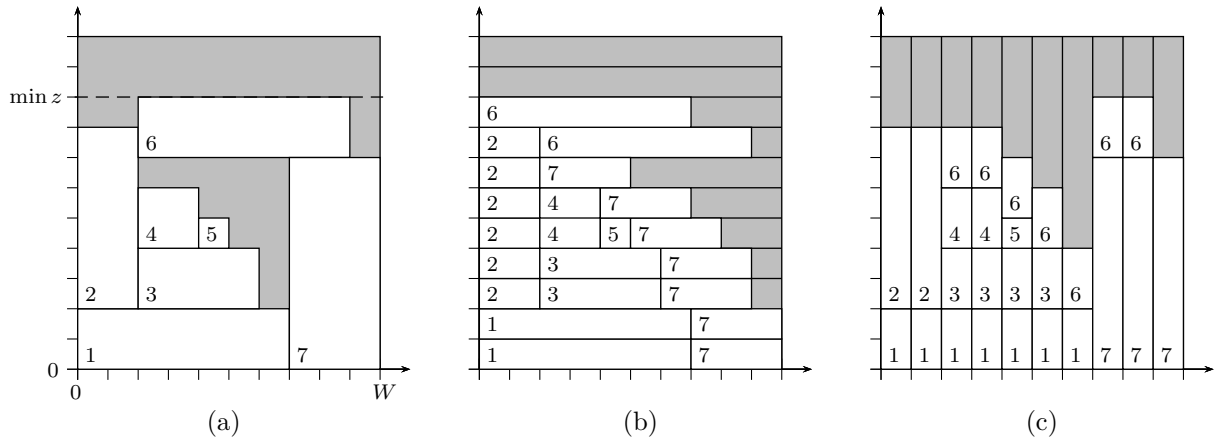

Figure 1: (a) an optimal SPP solution; (b) the 1CBP relaxation; (c) the $\mathrm{P}|cont|\mathrm{C}_{\max}$ relaxation.

The SPP is a challenging combinatorial problem. It is $\mathcal{NP}$-hard in the strong sense, and also very difficult to solve in practice. Benchmark instances proposed decades ago and containing just 20 items remain unsolved to proven optimality despite dozens of attempts. In terms of exact algorithms, the best results for the SPP have been obtained by the use of combinatorial branch-and-bound algorithms that build solutions by packing items one at a time in the strip. Among these, we cite the algorithms by Martello et al. (2003), Lesh et al. (2004), Bekrar et al. (2007), Alvarez-

Valdes et al. (2009), Kenmochi et al. (2009), Boschetti and Montaletti (2010), and Arahori et al. (2012, forthcoming). Techniques based on other concepts have also been developed: *Mixed integer linear programs* (MILP) were used in Sawaya and Grossmann (2005), Westerlund et al. (2007), and Castro and Oliveira (2011), whereas SAT-based algorithms were developed in Grandcolas and Pinto (2010) and Soh et al. (2010).

In terms of approximation schemes, Harren et al. (2011) presented a $5/3 + \varepsilon$ polynomial time approximation scheme. Kenyon and Rémila (2000) proposed an asymptotic fully polynomial time approximation scheme providing a solution of cost not higher than $(1+\varepsilon)opt + O(1/\varepsilon^2)$, where *opt* is the optimal solution value. Jansen and Solis-Oba (2009) presented an asymptotic polynomial time approximation scheme giving solutions bounded by $(1+\varepsilon)opt + 1$.

For what concerns heuristic algorithms with good practical computational performance, almost all metaheuristic paradigms have been applied to the SPP. In recent years, good results have been obtained with a squeaky wheel optimization methodology in Burke et al. (2011), a two-stage heuristic in Leung et al. (2011), and a skyline heuristic in Wei et al. (2011).

The majority of the exact algorithms for the SPP make use of the two following relaxations, obtained by "cutting" items into unit-width or unit-height slices, respectively. The first relaxation is based on the well-known *bin packing problem*, in which a set of weighted items has to be packed into the minimum number of capacitated bins.

Definition 1 *The* Bin Packing Problem with Contiguity Constraints *(1CBP) is the relaxation of the SPP obtained by cutting each item j into h_j* slices *of height 1 and width w_j, and the strip into* bins *of height 1 and width W. The aim is to pack the slices into the minimum number of bins, by ensuring that slices derived from the same item are* contiguous *one to the other: If the $k-th$ slice of item j ($j \in \mathcal{N}, k = 1, 2, \ldots, h_j$) is packed in bin i, then the $(k+1)-th$ slice, if any, must be packed in bin $i+1$.*

The second relaxation is based on the standard *parallel processor scheduling problem* ($\mathrm{P}||\mathrm{C}_{\max}$), in which a set of jobs having a given *processing time* has to be scheduled on a set of processors, so as to minimize the largest total processing time assigned to a processor (*makespan*).

Definition 2 *The* Parallel Processor Scheduling Problem with Contiguity Constraints *($P|\mathrm{cont}|C_{\max}$) is the relaxation of the SPP obtained by cutting each item j into w_j* slices *of width 1 and height h_j, and the strip into W vertical slices. We associate each item slice to a job having processing time h_j, and each vertical strip slice to a processor. The aim is to assign the slices to the processors, by minimizing the makespan and ensuring contiguity between the slices of the same item (if the $k-th$ slice of item*

j is assigned to processor i, then the $(k+1)-th$ slice, if any, must be assigned to processor $i+1$).

Figure 1-(b) shows an optimal solution for the 1CBP relaxation of the SPP instance of Figure 1-(a): Items have been cut horizontally and packed in 9 bins. Similarly, Figure 1-(c) gives an optimal solution to the relaxation induced by the $\mathrm{P}|cont|\mathrm{C}_{\max}$ on the same SPP instance: Items have been cut vertically and assigned to the W processors using minimum makespan 9. (Note that, in the literature, most graphical representations of bin packing and parallel processing scheduling problems, when not related to the SPP, draw bins as vertical containers and processors as horizontal lines, so they are "90° rotated" with respect to our figures).

Both the 1CBP and the $\mathrm{P}|cont|\mathrm{C}_{\max}$ are known to be strongly $\mathcal{NP}-$hard. From a practical point of view they are, however, easier than the SPP, and the solution values they provide (that can be different one from the other) are usually tight lower bounds on the optimal SPP height.

Several algorithms, starting from Martello et al. (2003), also used the 1CBP and/or the $\mathrm{P}|cont|\mathrm{C}_{\max}$ solution to try to compute a feasible solution for the original SPP instance. Suppose we are given the $\mathrm{P}|cont|\mathrm{C}_{\max}$ solution of Figure 1-(c), we can try to obtain the feasible SPP solution of Figure 1-(a) by adjusting the $y-$coordinates of the slices, so that all slices belonging to the same item are at the same $y-$coordinate, always ensuring that there is no overlapping among items. This problem can be defined as follows.

Definition 3 *Given a feasible $P|\mathrm{cont}|C_{\max}$ solution using makespan z, in which the first slice of an item $j \in \mathcal{N}$ is packed in processor x_j, problem $y-$check is to determine if there exists an array y_j, with $0 \leq y_j \leq z - h_j$, and such that the solution in which any item j is packed with its bottom left corner in position (x_j, y_j) is feasible for the SPP.*

Problem $y-$check is strongly $\mathcal{NP}-$complete (this is proved in Section 3.1), and most of the attempts developed in the literature for its solution are heuristics. The approach we propose is very innovative with respect to the literature, because we solve, instead, problem $y-$check with an exact algorithm, and, most important, we use it in a systematic way to optimally solve the SPP.

In particular, we propose a new exact algorithm for the SPP that exploits the full potentiality of the introduced relaxations by means of a Benders' decomposition. At the first step, in the master problem, we solve to optimality the $\mathrm{P}|cont|\mathrm{C}_{\max}$ relaxation. Then we try to obtain a feasible SPP solution, by solving the slave problem $y-$check. If a feasible solution is achieved, then it is also optimal, and hence we terminate. Otherwise a *Benders' cut* prohibiting the current $\mathrm{P}|cont|\mathrm{C}_{\max}$ solution is added to the master, and the procedure is re-iterated.

Benders' cuts are known to be weak in practice, and hence we try to strengthen them by borrowing the concept of *combinatorial Benders' cuts*, introduced in Codato and Fischetti (2006). In practice we look for a *minimal infeasible subset*, i.e., a minimal subset of items that still causes the infeasibility of the considered solution, and introduce the cut only for this subset, instead than for the complete set of items. After this has been done, we further improve the cut by means of a tailored lifting procedure based on the solution of linear programs.

The three problems we address (master, slave and search for the minimal infeasible subset) are all difficult, and possibly need to be solved several times. However, our resulting algorithm is usually fast in practice and, also due to a large number of optimization techniques that we propose, obtains very good computational results on the benchmark sets of instances.

1.1 Main Contributions of this Paper

The main contributions of this work are the following:

- we propose an innovative Benders' decomposition that models the SPP and exploits the full potentiality of the $\mathrm{P}|cont|\mathrm{C}_{\max}$ relaxation;
- we present several pre-processing, lower bounding, heuristic and exact algorithms for the master problem (derived from the $\mathrm{P}|cont|\mathrm{C}_{\max}$), taken from the related literature or newly developed;
- we prove that the slave problem in our decomposition (problem $y-$check) is strongly $\mathcal{NP}$-hard;
- we solve problem $y-$check with an algorithm based on new preprocessing techniques and a new enumeration tree enriched with fathoming criteria, and show that this method is highly efficient in practice;
- we propose non trivial ways to strengthen the Benders' cuts into combinatorial Benders' cuts, and present a new effective lifting procedure based on the solution of a linear model;
- we design an overall algorithm for the SPP, and test it on the benchmark instances obtaining very good computational results. In particular, we solve for the first time to proven optimality instances cgcut03 by Christofides and Whitlock (1977), and instances gcut04 and gcut11 by Beasley (1985). We provide 34 new proven optimal solutions for the 500 instances proposed by Berkey and Wang (1987) and Martello and Vigo (1998). We obtain, on average, better

solutions than all previously published exact algorithms, with a comparable or smaller computational effort.

2 A Benders' Decomposition for the Strip Packing Problem

We first provide the necessary notation, and then describe our decomposition approach and the prior work in the related literature.

2.1 Notation

We suppose the strip is located in the positive quadrant of the Cartesian coordinate system, with its bottom left corner located in position (0,0), as shown in Figure 1-(a). Let H be a valid upper bound on the optimal solution value. For simplicity we call *rows* the H unit-height bins obtained by cutting the strip horizontally (see Figure 1-(b)), and *columns* the W unit-width processors obtained by cutting the strip vertically (see Figure 1-(c)). Rows are numbered from 0 to $H-1$, and columns from 0 to $W-1$. We say that an item *covers* a row, resp. a column, if the row, resp. the column, intersects the item in the considered packing (e.g., item 3 in Figure 1-(a) covers rows 2 and 3, and columns 2, 3, 4, and 5).

We say that an item j is packed in position (p_j, r_j) if its bottom left corner has $x-$coordinate equal to p_j and $y-$coordinate equal to r_j (e.g., item 3 in Figure 1-(a) is packed in (2,2)). For feasibility we have that $0 \leq p_j \leq W - w_j$ and $0 \leq r_j \leq H - h_j$. This set of feasible positions may be reduced by considering the well-known principle of *normal patterns* by Herz (1972) and Christofides and Whitlock (1977), which states that there is an optimal solution in which each item is moved as down and as left as possible (hence touching at its left, and at its bottom, either the strip or the border of another item). To this aim we define

$$\mathcal{W}(j) = \left\{ p_j = \textstyle\sum_{i \in \mathcal{N} \setminus \{j\}} w_i \xi_i : 0 \leq p_j \leq W - w_j, \xi_i \in \{0,1\}\ \forall i \in \mathcal{N} \setminus \{j\} \right\}, \quad (1)$$

$$\mathcal{H}(j) = \left\{ r_j = \textstyle\sum_{i \in \mathcal{N} \setminus \{j\}} h_i \xi_i : 0 \leq r_j \leq H - h_j, \xi_i \in \{0,1\}\ \forall i \in \mathcal{N} \setminus \{j\} \right\} \quad (2)$$

the sets of normal patterns for item j along the $x-$ and $y-$axis, respectively. Sets $\mathcal{W}(j)$ and $\mathcal{H}(j)$ are computed using a standard dynamic programming technique, see, e.g., Christofides and Whitlock (1977). We similarly define

$$\mathcal{W}(j,q) = \{ p_j \in \mathcal{W}(j) : q - w_j + 1 \leq p_j \leq q \}, \quad (3)$$

$$\mathcal{H}(j,t) = \{ r_j \in \mathcal{H}(j) : t - h_j + 1 \leq r_j \leq t \} \quad (4)$$

the subset of normal patterns along the $x-$axis for which item j occupies column q, and the subset of normal patterns along the $y-$axis for which item j occupies row t, respectively. Finally, let $\mathcal{W} = \bigcup_{j\in\mathcal{N}} \mathcal{W}(j)$ and $\mathcal{H} = \bigcup_{j\in\mathcal{N}} \mathcal{H}(j)$ be the global sets of normal patterns along the $x-$ and $y-$axis, respectively.

2.2 A Mathematical-Logic Model

The SPP can be modeled by using two sets of variables: A binary variable x_{jp} taking value 1 if item j is packed in column p, 0 otherwise, and a continuous non-negative variable y_j giving the height of the bottom border of item j. A single variable z is then used to define the total height of the solution. The SPP can be described through the following mathematical-logic model:

$$(\text{SPP}_0) \qquad \min\ z, \tag{5}$$

$$\sum_{p\in\mathcal{W}(j)} x_{jp} = 1 \qquad j \in \mathcal{N}, \tag{6}$$

$$\sum_{j\in\mathcal{N}} \sum_{p\in\mathcal{W}(j,q)} h_j x_{jp} \le z \qquad q \in \mathcal{W}, \tag{7}$$

$$y_j + h_j \le z \qquad j \in \mathcal{N}, \tag{8}$$

$$\text{non-overlap}\{[y_j, y_j + h_j], j \in \mathcal{N} : \sum_{p\in\mathcal{W}(j,q)} x_{jp} = 1\} \qquad q \in \mathcal{W}, \tag{9}$$

$$x_{jp} \in \{0,1\} \qquad j \in \mathcal{N}, p \in \mathcal{W}(j), \tag{10}$$

$$y_j \ge 0 \qquad j \in \mathcal{N}. \tag{11}$$

Constraints (6) impose that each item is packed in exactly one column. Constraints (7) force z to be not smaller than the total height of the items that occupy any column q, whereas constraints (8) force z to be smaller than the upper border of any item j. Note that constraints (7) are not strictly necessary for the correctness of the model, but are essential for our decomposition approach. Logical constraints (9) impose that the vertical intervals $[y_j, y_j + h_j]$ corresponding to the set of items that occupies the same column q, do not overlap.

2.3 The Decomposition Approach

In the classical decomposition approach by Benders (1962), the aim is to solve a MILP problem P: $\min\{c^T y + f(x) : Ay + g(x) \ge b, y \ge 0, x \in D_x\}$. The method starts by finding a vector $\overline{x} \in D_x$, and considers the linear *slave problem* SP: $\min\{c^T y + f(\overline{x}) : Ay + g(\overline{x}) \ge b, y \ge 0\}$, which can be solved by means of the *dual slave* SD: $\max\{u^T(b - g(\overline{x})) + f(\overline{x}) : u^T A \le c, u \ge 0\}$. A solution $\overline{u}$ of SD induces a linear

constraint $z \geq \overline{u}^T(b-g(x))+f(x)$, the so-called Benders' cut, that is used to populate the *master problem* MP: $\min\{z : z \geq u_k^T(b-g(x))+f(x), k=1,2,\ldots,K, x \in D_x\}$, where $u_1, u_2, \ldots, u_K$ are the solutions of K dual problems obtained by iterating the above procedure.

A special case occurs when $c = 0$ and we start by optimally solving the master problem obtained by removing variables y from P, i.e., by setting $\overline{x} = \text{argmin}\{f(x) : g(x) \geq b, x \in D_x\}$. The slave SP then becomes a feasibility check on the system $\{Ay + g(\overline{x}) \geq b, y \geq 0\}$. If SP has a solution $\overline{y}$, then $(\overline{x}, \overline{y})$ is an optimal solution to P. If, instead, SP has no feasible solution, then $\overline{x}$ is not feasible for P and we know that at least one of the x_j variables must take a value different from $\overline{x}_j$. We write this condition as a linear constraint and add it to the master problem.

A better implementation does not add the cut containing all the x variables, but finds a smaller (possibly minimal) subset of variables that still induces infeasibility in the slave problem, and uses this set to derive the cut. The resulting constraint is called *combinatorial Benders' Cut* in Codato and Fischetti (2006), but the method can also be seen as an implementation of the *Logic-based Benders' decomposition approach* presented in Hooker (2000), Jain and Grossmann (2001), and Hooker and Ottosson (2003). We finally observe that, in the case of combinatorial Benders' cuts, it is not necessary that the slave is a continuous linear model.

For the SPP, when we remove variables y from model (5)–(11), we obtain a MILP that models the P$|cont|$C$_{\max}$ problem, namely:

$$(\text{P}|cont|\text{C}_{\max}) \qquad \min \; z, \tag{12}$$

$$\sum_{p \in \mathcal{W}(j)} x_{jp} = 1 \qquad j \in \mathcal{N}, \tag{13}$$

$$\sum_{j \in \mathcal{N}} \sum_{p \in \mathcal{W}(j,q)} h_j x_{jp} \leq z \qquad q \in \mathcal{W}, \tag{14}$$

$$x_{jp} \in \{0,1\} \qquad j \in \mathcal{N}, p \in \mathcal{W}(j). \tag{15}$$

Suppose now an integer solution $\mathcal{S} = \{z^s, x^s_{jp}\}$ to (12)–(15) has been computed: the slave is then to find a feasible solution, if any, to problem

$$(y-\text{check}) \qquad y_j + h_j \leq z^s \qquad j \in \mathcal{N}, \tag{16}$$

$$\text{non-overlap}\{[y_j, y_j + h_j], j \in \mathcal{N} : \sum_{p \in \mathcal{W}(j,q)} x^s_{jp} = 1\} \qquad q \in \mathcal{W}, \tag{17}$$

$$y_j \geq 0 \qquad j \in \mathcal{N}. \tag{18}$$

If $y-$check returns a feasible solution, then we obtained an optimal solution of the original SPP instance. Otherwise we forbid the current P$|cont|$C$_{\max}$ solution $\mathcal{S}$, by

adding a cut to the master problem. To this aim, let

$$p_j^s = \sum_{p \in \mathcal{W}(j)} p \; x_{jp}^s \tag{19}$$

denote the $x-$coordinate of the first slice of item j in solution $\mathcal{S}$. The Benders' cut is then

$$\sum_{j \in \mathcal{N}} x_{j,p_j^s} \leq n - 1. \tag{20}$$

Suppose now we can find a reduced subset of items $C^s \subseteq \mathcal{N}$, such that, if we pack all its items in position p_j^s, then problem $y-$check still has infeasible solution (the way in which we look for C^s is discussed in Section 4). Then, we obtain the *combinatorial Benders' cut*

$$\sum_{j \in C^s} x_{j,p_j^s} \leq |C^s| - 1. \tag{21}$$

We can then model the SPP as the following master problem:

$$\text{(SPP)} \qquad \min \; z, \tag{22}$$

$$\sum_{p \in \mathcal{W}(j)} x_{jp} = 1 \qquad j \in \mathcal{N}, \tag{23}$$

$$\sum_{j \in \mathcal{N}} \sum_{p \in \mathcal{W}(j,q)} h_j x_{jp} \leq z \qquad q \in \mathcal{W}, \tag{24}$$

$$\sum_{j \in C^s} x_{j,p_j^s} \leq |C^s| - 1 \qquad \forall \; C^s \text{ infeasible to } y-\text{check}, \tag{25}$$

$$x_{jp} \in \{0,1\} \qquad j \in \mathcal{N}, p \in \mathcal{W}(j). \tag{26}$$

The good aspect of this decomposition is the fact that it allows to develop tailored optimization techniques both for the master and the slave, taking advantage of their combinatorial structures.

In particular, we solve the slave with preprocessing techniques and an enumeration tree enriched with fathoming criteria, obtaining an algorithm (see Section 3) that is very fast in practice. For the master, we found computationally convenient to develop an iterative procedure that attempts different tentative strip heights, in an interval given by valid lower and upper bounds. At each attempt it solves the recognition version of the master, and updates the tentative strip height accordingly. The procedure is described in details in Section 5, and is based on preprocessing techniques, a large set of lower and upper bounding algorithms, and the direct solution of the MILP (22)–(26) with delayed cut generation.

The latter algorithm largely benefits from techniques aimed at finding improved Benders' cuts, that we describe in Section 4. The literature on this area of research is quite new, so we briefly summarize it in the next section.

2.4 Prior Work

The concept of primal decomposition of a MILP was originally proposed by Benders (1962), who studied the case in which the master results in a MILP and the slave in a LP. Later, Geoffrion (1972) generalized it to the case in which also the slave is a MILP.

In recent years, Hooker and Ottosson (2003) presented the concept of *logic-based Benders' decomposition*, a general framework in which both master and slave are MILPs, and the slave is solved by logical deduction methods, whose outcome is used to produce valid cuts. An interesting use of this approach is the one in which the master is solved by using standard MILP optimization, and the slave with a *Constraint Program* (CP). Successful examples of this type of decomposition have been proposed by, e.g., Jain and Grossmann (2001) and Hooker (2007). Jain and Grossmann (2001) present hybrid MILP/CP decomposition methods to solve a class of problems where the variables in the slave have zero coefficient in the original objective function, and apply them to the problem of scheduling jobs on parallel machines with release and due dates, while minimizing the sum of input processing costs. The master is a MILP that produces an assignment of jobs to machines and is solved with IBM Ilog Cplex. The slave is a CP that checks the feasibility of each assignment of jobs to a machine, and is solved with IBM Ilog Scheduler. Hooker (2007) proposes a similar approach to solve again a class of scheduling problems on parallel machines, but with the aim of minimizing either cost, makespan, or total tardiness.

Note that our algorithm differs from the ones by Jain and Grossmann (2001) and Hooker (2007), because it solves master and slave with dedicated combinatorial algorithms, and considers the more general case in which the activities of the machines are not independent one from the other, but strictly related among them (intuitively, an item j must be assigned to w_j machines).

The name "combinatorial Benders' cuts" was introduced by Codato and Fischetti (2006), who studied a decomposition in which the master is an ILP involving binary variables x, and the slave is a LP involving continuous variables y. Whenever a solution to the master is infeasible for the slave, they look for a *minimal infeasible subsystem* (MIS) of the LP associated to the slave, and then introduce in the master the corresponding cut. Since the problem of determining a MIS is $\mathcal{NP}$-hard, they make use of a greedy algorithm. Moreover, they limit their study to the case in which the constraints relating x and y are given by linear inequalities, each containing a single entry for the x array. Our approach differs from the one by Codato and Fischetti (2006) in several aspects, the most important being the fact that the slave is not a LP, but a strongly $\mathcal{NP}-$complete problem.

3 Solution of the Slave Problem

We are given the input of the SPP, plus a tentative strip height z^s and a vector p_j^s that gives the $x-$coordinate (i.e., column) in which item j is packed. In our decomposition vector p_j^s is obtained from a starting solution $\mathcal{S} = \{z^s, x_{jp}^s\}$ to (12)–(15), by using (19). In this section we describe how to solve the resulting $y-$check problem, see Definition 3, which calls for the determination of the $y-$coordinates to be assigned to each item so as to obtain a feasible SPP solution of height z^s, if any exists. We first discuss the problem complexity, and then present our solution algorithm.

3.1 Complexity

Let us consider the SPP solution depicted in Figure 2, where 9 items are packed in a strip of height $z^s = 2B + 3$, with B a given integer positive value. All items have width 1, with the exception of items 3, 4, and 5, that have width 3. They are packed in the $x-$coordinates $p^s = [0, 4, 0, 1, 2, 1, 3, 3, 1]$, and have heights $h = [2B + 2, 2B + 2, 1, 1, 1, B + 1, B + 1, B, B]$.

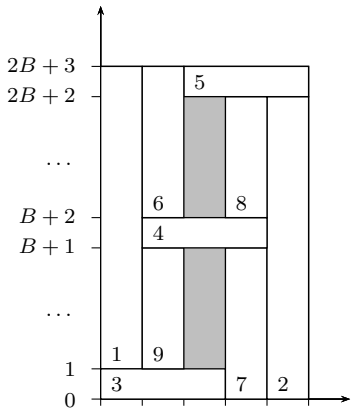

Figure 2: A framework to reduce PARTITION to $y-$check.

Items 3 and 5 cannot be packed at the same y-coordinate, because they would overlap, so, due to the presence of items 1 and 2, item 3 must be assigned at the bottom of the strip and item 5 at the top (the obvious symmetric solution where 3 is at the top and 5 at the bottom is also possible). A further examination of the remaining items shows that the only feasible $y-$check solution for items 4, 6, 7, 8, and 9 is the one depicted in the figure. This solutions leaves two empty *buckets* of width 1 and height B. These buckets can be used to prove the $\mathcal{NP}-$completeness of $y-$check, by using a transformation from the following problem.

Definition 4 PARTITION: *Given $\overline{n}$ items, each having weight $s_j \in Z^+$ $(j = 1, 2, \ldots, \overline{n})$, find a subset $S \subseteq \{1, 2, \ldots, \overline{n}\}$ such that $\sum_{j \in S} s_j = \sum_{j=1}^{\overline{n}} s_j/2$, if any.*

Lemma 1 *Problem $y-$check is $\mathcal{N}P-$complete.*

Proof We construct an instance of $y-$check with $n = \overline{n} + 9$ items. The first 9 items are those depicted in Figure 2, with $B = \sum_{j=1}^{\overline{n}} s_j/2$. The remaining items have $p_j^s = 2$, $w_j = 1$ and $h_j = s_{j-9}$, for $j = 10, 11, \ldots, n$. All the items $10, 11, \ldots, n$ have to be packed inside the two buckets, so $y-$check has a feasible solution if and only if PARTITION has. Since PARTITION is $\mathcal{N}P-$complete, the same holds for $y-$check. □

By using an extension of the above Lemma, we can prove the following stronger result.

Theorem 1 *Problem $y-$check is* strongly *$\mathcal{N}P-$complete.*

The proof is given in Appendix 8.1.

3.2 Algorithms for Problem $y-$check

For the solution of $y-$check we developed several algorithms, and obtained the best computational performance by using a combinatorial enumeration tree, enriched by reduction and fathoming criteria. The resulting algorithm, called $y-$check algorithm in the remaining of the paper, starts with the three new preprocessing techniques, invoked in sequence one after the other.

Preprocessing 1: Merge items.

For any item j, let us define $\mathcal{L}(j)$, resp. $\mathcal{R}(j)$, the subset of items that can be packed at the left, resp. right, of j. Formally,

$$\mathcal{L}(j) = \{i \in \mathcal{N} \setminus \{j\} : p_i^s + w_i \leq p_j^s\}, \tag{27}$$

$$\mathcal{R}(j) = \{i \in \mathcal{N} \setminus \{j\} : p_i^s \geq p_j^s + w_j\}. \tag{28}$$

We consider items one at a time, for non-increasing order of p_j^s. For a given item j, if $h_i \leq h_j$ holds for all $i \in \mathcal{L}(j)$, then we attempt packing the items of $\mathcal{L}(j)$ in the substrip of width p_j^s and height h_j, by invoking the $y-$check enumeration tree described at the end of this section. If all items in $\mathcal{L}(j)$ fit into the substrip, then we merge j and $\mathcal{L}(j)$ into a unique item, say k, having $w_k = p_j^s + w_j$, height $h_k = h_j$ and $p_k^s = 0$. This preserves the optimality of the solution, because no other item can enter the induced substrip.

If not all the items in $\mathcal{L}(j)$ fit into the substrip, or there are some items i having $h_i > h_j$, then we remove items from $\mathcal{L}(j)$ in an iterative way. We proceed from left to

right: Let $\bar{p}$ be the first column occupied by an item in $\mathcal{L}(j)$ and $\bar{w}$ be the largest width of an item in $\mathcal{L}(j)$ being packed in $\bar{p}$. We check if $\mathcal{L}(j)$ can be exactly partitioned into two subsets, one completely contained in the columns $[0, 1, \ldots, \bar{p} + \bar{w} - 1]$, and one completely contained in the columns $[\bar{p} + \bar{w}, \bar{p} + \bar{w} + 1, \ldots, p_j^s - 1]$. If this is possible, then we focus our search on the latter group of columns. Formally, we check if there are no items $i \in \mathcal{L}(j)$ having $p_i^s < \bar{p} + \bar{w}$ and $p_i^s + w_i > \bar{p} + \bar{w}$. If no items with this property exist, then we set $\mathcal{L}(j) = \{i \in \mathcal{N} \setminus \{j\} : \bar{p} + \bar{w} < p_i^s + w_i \leq p_j^s\}$. In this way, we are left with a reduced set of items and a reduced substrip of width $p_j^s - (\bar{p} + \bar{w})$ and height h_j. Once again, no item outside $\mathcal{L}(j)$ may enter this reduced substrip at the left of j, and hence we invoke the enumeration tree to try to merge j and the reduced set $\mathcal{L}(j)$. If instead $\mathcal{L}(j)$ cannot be partitioned, then we increase $\bar{p}$ to be the next column where an item of $\mathcal{L}(j)$ is packed, and re-attempt the partition.

We re-iterate until a merging is obtained or $\mathcal{L}(j)$ is empty. We then repeat the process with $\mathcal{R}(j)$, for which we iterate from right to left. For an example, consider Figure 1-(c) and $j = 4$. At the left, $\mathcal{L}(4) = \{2\}$ and no merging is possible. At the right, $\mathcal{R}(4) = \{5, 7\}$ at the first iteration and no merging is possible. At the second iteration $\mathcal{R}(4) = \{5\}$, and 4 and 5 are merged into a unique item of width 3 and height 2.

Preprocessing 2: Lift item widths.

We consider items one at a time, by non-decreasing width, breaking ties by non-decreasing height. For any item j, we compute $\mathcal{L}(j)$ and $\mathcal{R}(j)$ using (27) and (28), respectively. Let $\ell_j = \max_{i \in \mathcal{L}(j)}\{p_i + w_i\}$ if $\mathcal{L}(j)$ is not empty, and $\ell_j = 0$ otherwise. Similarly, let $r_j = \min_{i \in \mathcal{R}(j)}\{p_i\}$ if $\mathcal{R}(j)$ is not empty, and $r_j = W$ otherwise. We move item j to its left as much as possible, by setting $p_j^s = \ell_j$, and then enlarge its width as much as possible, by setting $w_j = r_j - \ell_j$. We then re-iterate with the next item.

Note that this preserves the optimality of the solution, because no item can be packed side-by-side with j in the columns between ℓ_j and r_j. Consider again Figure 1-(c). This second preprocessing would produce $w_3 = 5$, $w_4 = 5$ (recall items 4 and 5 were merged by the previous preprocessing) and $w_6 = 8$. The outcome is depicted in Figure 3-(a).

Preprocessing 3: Shrink the strip.

This technique is based on the following simple idea. Suppose that a column p is occupied by a set S of items, and that a feasible packing for these items exists. A consequence is that, if no item outside S occupies column $p + 1$, then the packing of the items in S is also feasible for column $p + 1$. In practice, it is enough to check the

feasibility only for those columns where the left border of an item is packed. Thus we remove all other columns from the instance and reduce the widths of the items and of the strip accordingly. Consider for example the instance of Figure 3-(a). The only columns that we keep are column 0 ($p_1^s = p_2^s = 0$), column 2 ($p_3^s = p_4^s = p_6^s = 2$), and column 7 ($p_7^s = 7$). The instance that is given to the successive $y-$check enumeration tree in depicted in Figure 3-(b).

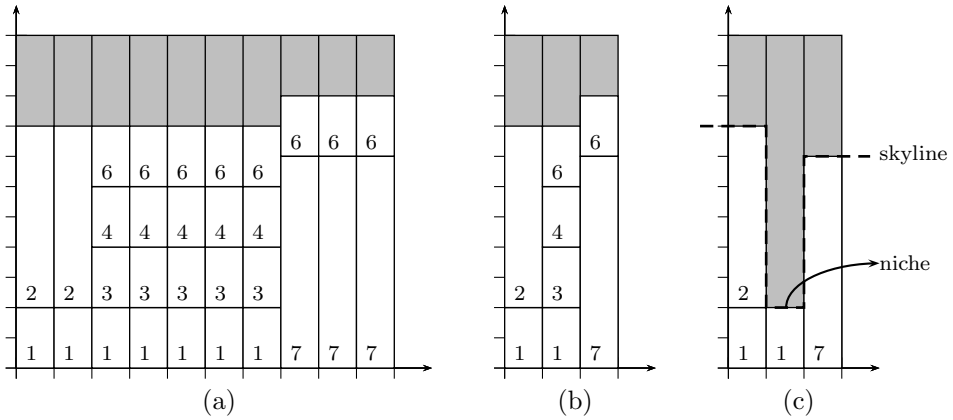

Figure 3: (a) instance of Figure 1-(c) modified by preprocessing 1 and 2; (b) further modification by preprocessing 3; (c) a partial packing.

Note that these $y-$check preprocessing techniques consistently reduce the size of the addressed instances, and are much more effective that the standard techniques for the SPP (see, e.g., Boschetti and Montaletti 2010, and references therein), because largely benefit from the additional information given by the vector p_j^s.

At the end of the preprocessing phase, all items having width equal to the width of the strip are packed at the bottom of the strip, and then removed from the instance. The packing of the remaining items is attempted with the following exact algorithm.

Enumeration tree for problem $y-$check.

This procedure constructs partial solutions by adding one item at a time, starting from an empty solution. For the sake of simplicity we continue using the standard notation adopted so far, i.e, n, W, H, w_j, h_j, ..., but recall that these values may have been modified by preprocessing. Moreover, let $h_{used}(p)$ be the sum of the heights of the items already packed at a given node, and covering column p, for $p = 0, 1, \ldots, W-1$.

Following a notation common in two-dimensional packing, we define the *skyline* as the line that touches the top of the packed items, see the dashed line in Figure 3-(c). In each column p the height of the skyline is given by $h_{used}(p)$. Let us also define the

niche as the horizontal segment of the skyline that has the smallest value of h_{used}, see again Figure 3-(c). If more horizontal segments having the same smallest value of h_{used} exist, then the niche is the leftmost one. In the following we suppose that the niche starts in column ℓ and ends in column r (with r included in the niche).

We define h_{left} and h_{right} as the heights of the left and right border of the niche, respectively, and compute them as follows. If $\ell = 0$ then $h_{left} = z^s$, else $h_{left} = h_{used}(\ell - 1)$. If $r = W - 1$ then $h_{right} = z^s$, else $h_{right} = h_{used}(r + 1)$. For example, in Figure 3-(b) we have $\ell = r = 1$, $h_{left} = 8$ and $h_{right} = 7$. Let $\mathcal{N}' \subseteq \mathcal{N}$ be the set of items still to be packed at a given node, and $\mathcal{N}'(\ell, r) \subseteq \mathcal{N}'$ the set of items that can be packed in the niche, i.e., $\mathcal{N}'(\ell, r) = \{j \in \mathcal{N}' : p_j^s \geq \ell \text{ and } p_j^s + w_j \leq r + 1\}$.

The enumeration tree has one node for each partial solution and branches on the items that can be packed in the associated niche. At the root node the strip is empty and the niche corresponds to the whole strip. We sort items in $\mathcal{N}'(\ell, r)$ by non-decreasing value of p_j^s, and create $|\mathcal{N}'(\ell, r)| + 1$ nodes. The first $|\mathcal{N}'(\ell, r)|$ nodes are created by selecting an item $j \in \mathcal{N}'(\ell, r)$, in order, and packing it in position p_j^s. The last node is obtained by packing no item at all in the niche, and, in this case, we close the whole niche and lift the skyline by setting $h_{used}(p) = \min\{h_{left}, h_{right}\}$ for $p = \ell, \ell + 1, \ldots, r$. When an item j is packed, if $p_j^s > \ell$, then we close the rectangular space of height h_j, starting in ℓ and terminating in $p_j^s - 1$, by setting $h_{used}(p) = \min\{h_{left}, h_{used}(p) + h_j\}$ for all $p = \ell, \ell + 1, \ldots, p_j^s - 1$. This is done to kill symmetries, because the solutions in which an item was packed in this rectangular space were already explored by previous nodes, due to the sorting. The resulting tree is explored in depth-first search.

At each node the following fathoming criteria are used:

1. let $h_{pack}(p)$ be the total height of the items in $\mathcal{N}'$ that cover column p, for $p = 0, 1, \ldots W - 1$. If $h_{used}(p) + h_{pack}(p) > z^s$ holds for a certain column p, then the current node is fathomed;

2. if $\mathcal{N}'(\ell, r)$ contains a unique item, say j, such that $h_{used}(p_j^s) + h_j \leq \min\{h_{left}, h_{right}\}$, then we create a single descendant node by packing j in p_j^s, and skip the (dominated) node in which no item is packed in the niche;

3. if there are two items j and k in $\mathcal{N}'(\ell, r)$, with $j < k$, having $w_j = w_k$, $h_j = h_k$ and $p_j^s = p_k^s$, then we fathom those nodes that attempt packing k before j (because the same solution is found by the 'twin' node that packs first j and then k);

4. when packing item j, we check if there is an item k having $w_k = w_j$, $k > j$ and being already packed in p_j^s at height $h_{used}(p_j^s) - h_k$, i.e., at the top of the skyline. If this is the case, then the node is fathomed (again, the same solution

is found by the 'twin' node that packs first j and then k, note that in this case h_j can be different from h_k);

5. when packing item j, if $p_j^s > \ell$ holds, then we check if there exists an item $k \in \mathcal{N}'(\ell, r)$, $k \neq j$, that enters completely in the rectangle at the left of j that would be closed by the packing of j, i.e., an item k having $p_k^s \geq \ell$, $p_k^s + w_k \leq p_j^s$, and $h_k \leq \min\{h_{left} - h_{used}(\ell), h_j\}$. If such k exists, then we fathom the node (because the same solution is found by packing first k and then j).

4 Improving and Lifting the Benders' Cuts

In the following, we suppose we are given a solution $\mathcal{S} = \{z^s, x_{jp}^s\}$, which is infeasible to $y-$check and has to be cut from the master problem. To improve the standard Benders' cuts (20), we developed a procedure that uses four steps (all newly developed ideas). In the first three steps, described in Section 4.1, we look for *minimal infeasible subsets*, i.e., minimal subsets of items still producing an infeasible $y-$check instance, and use them to derive the stronger cuts (21). In the last step, described in Section 4.2, we try to further lift the cut by adding x_{jp} variables through the solution of linear models. In the following, recall that vector p^s gives the $x-$coordinates in which items are packed in $\mathcal{S}$.

4.1 Finding Minimal Infeasible Subsets of Items

The problem of determining a minimal infeasible subset is $\mathcal{NP}$–hard, because the underlying recognition problem, $y-$check, is $\mathcal{NP}$–complete, thus we content us to solve it in a greedy fashion.

The first step of our procedure looks for *vertical cuts* in the packing induced by $\mathcal{S}$, i.e., for columns $p \in \mathcal{W}$ such that $\mathcal{N}$ can be partitioned into two sets, $\mathcal{N}_1 = \{j \in \mathcal{N} : p_j^s + w_j \leq p\}$ and $\mathcal{N}_2 = \{j \in \mathcal{N} : p_j^s \geq p\}$, with $\mathcal{N}_1 \cup \mathcal{N}_2 = \mathcal{N}$. If such a column exists, then the packing of the items in $\mathcal{N}_1$ is not influenced by the packing of the items in $\mathcal{N}_2$, and viceversa. Thus, we re-execute the $y-$check algorithm on both $\mathcal{N}_1$ and $\mathcal{N}_2$ and determine which subset is infeasible (at least one is infeasible because $\mathcal{S}$ is so). Clearly, if k vertical cuts are found, the set of items is partitioned into $k+1$ subsets accordingly. Then, the successive steps of our procedure are executed on the resulting infeasible subset(s) of items.

The second step tries to remove one column at a time from the strip. It first removes from $\mathcal{S}$ column 0 and all items j having $p_j^s = 0$. If the reduced instance is infeasible for $y-$check , then it continues by removing the next column from the left in which at least one item is packed, and all items packed in that column. It

re-iterates as long as the reduced instance remains infeasible. Then, it starts from the right, by selecting column $W-1$ and all items that occupy it, removing them from the instance and invoking the $y-$check algorithm. Again, it re-iterates as long as the reduced instance remains infeasible.

The third step considers the items one at a time according to a given ordering. It removes the current item from $\mathcal{S}$ and re-executes the $y-$check algorithm on the reduced instance. If the outcome is feasible, then the item is re-inserted in $\mathcal{S}$, otherwise we found a reduced cause of infeasibility and keep the current item out of $\mathcal{S}$. In any case, we re-iterate with the next item until all items have been scanned. At the end of the scan, we are left with a reduced subset of items, that still induces an infeasibility. The result of this step depends on the order in which items are selected. For this reason we perform several attempts with different orderings. The first attempt selects items according to non-decreasing value of area. In the second attempt we assign a *success score* with each item. This is initially set to 0 at the beginning of the solution of the current SPP instance, and then increased by one unit if the removal of the item from $\mathcal{S}$ was successful in previous iterations of the decomposition algorithm (i.e., if it led to a reduced instance that was still infeasible). The second attempt we perform then selects items according to non-decreasing value of success score. The third attempt simply selects items randomly, and is executed ten times.

A simple hash list is used to keep track of the reduced instances for which the $y-$check algorithm was invoked, so as to avoid duplicate calls. At the end of these three steps we typically obtain a few minimal infeasible subsets of items, $C^s \subseteq \mathcal{N}$, that can be used for the cuts of type (21).

4.2 Lifting the Cut

The fourth and last step of our procedure aims at lifting (21), for a certain $C^s \subseteq \mathcal{N}$, as follows. For each item j, let us define $\mathcal{K}^s(j)$ the subset of items that vertically overlap with j in solution $\mathcal{S}$, i.e., the set of items that have at least one column in common with j in $\mathcal{S}$. Suppose now that we move j from p_j^s to a different position at its left or at its right, keeping all other items in their original position in $\mathcal{S}$. As long as $\mathcal{K}^s(j)$ remains the same, then we know that $\mathcal{S}$ is still infeasible for $y-$check. Without the need of supplementary calls to $y-$check, we can thus obtain a set of $x-$coordinates for the left border of j that keep $\mathcal{S}$ infeasible. In terms of binary variables, we can add to the left-hand side of the cut all the x_{jp} variables corresponding to the selected $x-$coordinates, without affecting the right-hand side (because just one of the coordinates can be selected for packing j), thus lifting the original cut.

To obtain the most effective lifting we proceed in the following way. For each item $j \in C^s$ we introduce two non-negative variables, l_j^s and r_j^s, and denote with $[l_j^s, r_j^s]$ the interval along the $x-$axis in which we look for the $x-$coordinate of j. Let also $\mathcal{K}_p^s(j)$

be the subset of items that vertically overlap with j, when j is packed in column $p \in [l_j^s, r_j^s]$. If $\mathcal{K}_p^s(j) = \mathcal{K}^s(j)$ for all $j \in C^s$, then problem $y-$check is still infeasible. We can thus find the largest lifting by solving the LP:

$$(\mathcal{C}^s\text{–lift}) \qquad \max \sum_{j \in \mathcal{C}^s} \left(r_j^s - l_j^s\right), \tag{29}$$

$$l_j^s + w_j \geq r_i^s + 1 \qquad j \in \mathcal{C}^s, i \in \mathcal{K}^s(j), \tag{30}$$

$$0 \leq l_j^s \leq p_j^s \qquad j \in \mathcal{C}^s, \tag{31}$$

$$p_j^s \leq r_j^s \leq W - w_j \qquad j \in \mathcal{C}^s. \tag{32}$$

Constraints (30) impose that items j and i overlap in any solution in which j is packed in any column of $[l_j^s, r_j^s]$ and i in any column of $[l_i^s, r_i^s]$ (to this aim, note that, if $i \in \mathcal{K}^s(j)$, then $j \in \mathcal{K}^s(i)$). Constraints (31) and (32) are used to state that, for any item j, the selected interval $[l_j^s, r_j^s]$ is such that (i) the item always lies inside the strip, and (ii) the original position p_j^s of the item in $\mathcal{S}$ is inside the interval.

We then use the optimal solution $(\bar{l}_j^s,\ \bar{r}_j^s)$ to (29)–(32), to obtain the following *lifted combinatorial Benders' cut*:

$$\sum_{j \in \mathcal{C}^s} \sum_{p \in [\bar{l}_j^s, \bar{r}_j^s]} x_{jp} \ \leq |\mathcal{C}^s| - 1. \tag{33}$$

Summarizing, starting from the original Benders' cut (20), we obtain several (much stronger) lifted combinatorial Benders' cuts (33) and add all of them to the master problem. The computational effectiveness of this procedure is shown in Section 6.

5 An Exact Algorithm for the Strip Packing Problem

The mathematical models and the algorithms presented in the previous sections have been inserted into an overall algorithm for the solution of the SPP. This algorithm also makes use of several additional techniques to speed up its convergence to the optimum, either best practises coming from the related literature, or newly developed procedures. For this reason we called it BLUE, from *Benders' decomposition with Lower and Upper bound Enhancements*.

An informal pseudo-code is outlined in Algorithm 1. Intuitively, BLUE first preprocesses the instance and computes an upper bound U and a lower bound L on the optimal solution value. Then, as long as L is strictly smaller than U, it solves the recognition version of the SPP, called SPP(L), in which the height of the strip is fixed to L, and the aim is to find a feasible packing of the items not exceeding L,

if any (the problem is also known in the literature as the *two-dimensional orthogonal packing problem*, see, e.g., Clautiaux et al. 2007). The SPP(L) instance is first passed to a preprocessing procedure, and then solved by two exact methods, namely a combinatorial branch-and-bound and the Benders' decomposition of Section 2. In the remaining of this section we give the details of each step of the algorithm.

Algorithm 1 Algorithm BLUE for the solution of the Strip Packing Problem

```
preprocess instance
compute an upper bound U and a lower bound L
while L < U do
    comment: Solve SPP(L) = recognition version of SPP at height L
    preprocess item heights
    if Σ_{j∈N} |H(j)| < Σ_{j∈N} |W(j)| then rotate instance
    solve SPP(L) with combinatorial branch-and-bound (B&B)
    if B&B failed then solve SPP(L) with Benders' decomposition
    if Benders' decomposition failed then return heuristic solution
    if SPP(L) is feasible then
        U := L
    else
        increase L
    end-if
end while
```

5.1 Preprocessing and Bounds

In the following we suppose items are sorted by non-increasing width, breaking ties by non-increasing height. Algorithm BLUE first preprocesses the instance using the three techniques described in Section 2.2 of Boschetti and Montaletti (2010). The first technique aims at packing large items at the bottom of the strip, and requires to run a heuristic (which in our case is the algorithm described below to compute U) on a subinstance. The second one aims at reducing the strip width W. The third one computes, for any item j, in order, the maximum total width w'_j of a subset of items that can be packed side by side with j without exceeding W, and then, if $w_j + w'_j < W$, it sets $w_j = W - w'_j$.

Lower bounds.

To obtain a valid lower bound we first make use of three polynomial-time procedures from the literature, namely:

1. The simple lower bound $L_1 = \max\{\lceil \sum_{j \in \mathcal{N}} w_j h_j / W \rceil; \max_{j \in \mathcal{N}} h_j\}$;

2. A more sophisticated lower bound, L_2, based on the most performing dual feasible functions. In practice, L_2 is evaluated as L^{BM}_{dff}, described in Section 3.2 of Boschetti and Montaletti (2010), but with the inclusion of only the first three dual feasible functions (the fourth one was disregarded because more time consuming and not computationally effective on our instances);

3. A third lower bound, L_3, obtained by invoking the alternating constructive procedure described in Section 4.2.6 of Alvarez-Valdes et al. (2009).

Then we invoke two newly developed and more time-consuming procedures. The first one is obtained by considering the relaxation of the 1CBP (see Definition 1), in which we remove the constraint that horizontal slices must be packed contiguously one with the other, and only impose that each bin contains at most one slice of each item. The problem, known in the literature as the *non-contiguous bin packing problem* (NCBP), can be modeled as follows.

We define a *pattern* t as a subset of items whose total width is not larger than W, and describe it by a column $(a_{1t}, \ldots, a_{jt}, \ldots, a_{nt})^T \in \{0,1\}^n$, where a_{jt} takes value 1 if item j is in pattern t, 0 otherwise. Let $\mathcal{T}$ be the family of all patterns containing at most one slice for each item, and let z_t be a binary variable taking value 1 if pattern t is used, 0 otherwise ($t \in \mathcal{T}$). The NCBP is then

$$\text{(NCBP)} \qquad \min \sum_{t \in \mathcal{T}} z_t, \tag{34}$$

$$\sum_{t \in \mathcal{T}} a_{jt} z_t \geq h_j \qquad j \in \mathcal{N}, \tag{35}$$

$$z_t \in \{0,1\} \qquad t \in \mathcal{T}. \tag{36}$$

Since the NCBP is strongly $\mathcal{NP}$-hard, we content us with the continuous relaxation of (34)–(36), which we solve by the standard column generation algorithm originally proposed by Gilmore and Gomory (1961) for the cutting stock problem. The fact that an item should appear at most once in a pattern is easily taken into account in the slave knapsack problem that has to be solved to generate columns. This is done by associating a binary variable with each item $j \in \mathcal{N}$ (instead of an integer variable not greater than h_j, as in the standard algorithm for the cutting stock). This column generation procedure is quite standard for one-dimensional packing problems, but, as far as we know, this is the first time it is applied to the solution of the NCBP.

Let L_4 be the round up of the resulting solution value. Now that we know that all horizontal slices can be continuously packed in L_4 bins, we check if the same fact holds for the vertical slices. We then fix the height of the strip to be L_4, "cut" items

and strip using the vertical orientation, and again solve the continuous relaxation of (34)–(36). In this second attempt patterns are combinations of items whose total height does not exceed L_4, and h_j is replaced by w_j in (35). If the resulting solution value is greater than W, then no feasible packing of the vertical slices exists, so we increase L_4 by one. We re-iterate this second attempt as long as its solution value is greater than W.

The last procedure is obtained by solving the root node of the MILP (12)–(15) for the P|*cont*|$\mathrm{C_{max}}$, and storing the rounded up value of the resulting makespan as L_5. The lower bound we obtain is $L = \max_{i \in 1..5}\{L_i\}$, where L_1, L_2 and L_3 have polynomial complexity, L_4 has pseudopolynomial complexity (if the ellipsoid algorithm is used to solve the LPs, see Caprara and Monaci 2009) and is fast in practice, whereas computing L_5 is strongly $\mathcal{NP}$-hard and can be time consuming for the large instances (this is why at this step we limit its solution to the root node).

Upper bounds.

To obtain a valid upper bound we start by invoking (our implementation of) the algorithm by Leung et al. (2011). This is a two-stage approach, in which the first stage is a constructive heuristic, and the second one is an improvement procedure based on simulated annealing. In our implementation we impose a limit of 10^4 iterations to the simulated annealing, to lower the computation time. We call U_1 the resulting solution value.

We then invoke a new heuristic based on the solution of the 1CBP (recall Figure 1-(b)). At each iteration this heuristic selects an unpacked item, and packs all its slices left-justified in the bins, starting from the bottom-most bin that has enough residual space to accommodate a slice. It re-iterates until all items are packed. Bins are closed once they cannot accommodate any more item. The resulting partial solutions have a classical staircase structure. In details, the first item j is selected randomly, and its slices are packed at the left of bins $0, 1, \ldots, h_j - 1$. Let r be the index of the bottom-most open bin. The next item is chosen by using a score-based method that has the aim of filling bin r in the best possible way. A *score* v_j is assigned to each item j which is still unpacked, with v_j initially set to 2 if j is as large as the residual space in r, 0 otherwise. Then, if by packing j in r, the top of j is as high as the first horizontal segment at its left in the staircase, then v_j is increased by 2. If instead the top of j is as high as the top of any other horizontal segment in the staircase but the first, then v_j is increased by 1. The item with highest score is selected (ties are broken randomly) and all its slices are packed.

After the heuristic packed all items, if the resulting 1CBP solution value is smaller than U_1, then we try to obtain a feasible SPP solution by invoking the $y-$check algorithm, with a limit of 3 CPU seconds or $2 \cdot 10^5$ iterations. The instances for which

$y-$check was invoked are stored in a hash table, so as to avoid checking them twice. The algorithm is executed 10 times, and the best solution value is stored in U_2. We then compute $U = \min\{U_1; U_2\}$.

Summarizing, the initialization of Algorithm BLUE computes in order U_1, L_1, L_2, L_3, L_4, L_5, and U_2, and updates U and L accordingly. The execution is clearly stopped whenever $L = U$.

5.2 Closing the Gap

After the preprocessing and the computation of the bounds, if L is strictly smaller than U, then we enter a loop in which we try to solve SPP(L). We first try to lift the item heights. To this aim we use the third preprocessing technique described in Section 5.1, but work on heights instead of widths. Formally, we compute for any item j in order the maximum total height h'_j of a subset of items that can be packed vertically over j without exceeding L, and then, if $h_j + h'_j < L$, we set $h_j = L - h'_j$.

As a general remark, it is usually "easier" (i.e., computationally faster) to solve a P$|cont|$C$_{\max}$ problem with a small number of columns, than a 1CBP with a large number of bins. For this reason we compute the sets of normal patterns along the $x-$ and $y-$axis, by using (1) and (2), respectively, and setting $H = L$. Then, if $\sum_{j\in\mathcal{N}} |\mathcal{H}(j)| < \sum_{j\in\mathcal{N}} |\mathcal{W}(j)|$ we "rotate" the instance, i.e., we exchange L with W, and h_j with w_j for all $j \in \mathcal{N}$. Clearly after the instance is solved we restore the original dimensions. Note that the two terms in the check give the number of variables in the resulting master problems with one orientation or the other, see (26).

We then invoke the two exact procedures for solving the SPP(L), both described in details below. If these procedures prove that a feasible solution at height L exists, then we stop the algorithm with a proof of optimality. If instead they prove that no feasible solution exists at height L, then we increase L and re-iterate. In the latter case, we compute the minimum integer $\ell > L$ such that there exists a feasible combination of item heights whose value is exactly ℓ, and then set $L = \ell$. If the two procedures fail in proving feasibility or infeasibility at height L, then the algorithm terminates by returning a heuristic solution.

Branch-and-Bound for the SPP(L).

The first attempt to find an exact SPP(L) solution is based on a branch-and-bound (B&B) for the recognition version of the P$|cont|$C$_{\max}$ in which the makespan is fixed to L. This B&B starts with W empty columns, and enumerates solutions by packing one item at a time in the left-most column that still has some residual space to accommodate items. Packing an item j in a column p means, in our approach,

packing all the slices of j at the bottom of columns $p, p+1, \ldots, p+w_j-1$. As a consequence, the resulting partial solutions have a staircase structure.

Let $\bar{h}_p$ be the total height of the slices packed in column p in a partial solution. At each node, the B&B first selects the left-most column p which still has $\bar{h}_p < L$. It then creates a descendant node for any item j satisfying $\bar{h}_p + h_j \leq L$. It packs j in p, computes $h_{\min}$ as the minimum height of the items still to be packed, and then sets $\bar{h}_q = \bar{h}_q + h_j$ if $\bar{h}_q + h_j + h_{\min} \leq L$, $\bar{h}_q = L$ otherwise, for $q = p, p+1, \ldots, p+w_j-1$. If column p is not empty (i.e., $\bar{h}_p > 0$ holds), then the B&B also creates an additional node in which it packs no item at all in p, and in this case it sets $\bar{h}_p = L$.

At any node of the tree we apply the following fathoming criteria:

1. if there are two items j and k, with $h_j = h_k$, $w_j = w_k$ and $j > k$, then we pack j only after k has been packed;

2. we define an array $i(p)$ in which we store, for any column p, the maximum index of an item packed with its left-most slice in p. At any iteration we allow to pack an item j in p only if $j > i(p)$;

3. we use the standard continuous lower bound on the area. Namely, if the area of the items still to be packed is greater than the residual area ($WL - \sum_{p=0,1,\ldots,W-1} \bar{h}_p$), then we fathom the node;

4. We make use of the *DP cut*, a lower bounding technique based on iterated solutions to subset sum problems, described in Section 4.3.1 of Kenmochi et al. (2009).

Criteria 1 and 2 are derived from Mesyagutov et al. (2011) and are used to kill symmetries: They forbid the same solution to be found at several nodes in the tree. Criteria 3 and 4 are standard lower bounding techniques that forbid infeasible SPP(L) solutions. Note that by using these criteria we are still enumerating all possible P$|cont|$C$_{\max}$ solutions, and hence we do not lose any possible solution for the SPP(L).

During the exploration of the B&B tree, whenever we find a feasible P$|cont|$C$_{\max}$ solution ("candidate" in the following), we invoke the $y-$check algorithm, with a maximum number of $2 \cdot 10^7$ iterations, to determine if it is also feasible for the SPP(L).

There are three possible exits for the B&B:

1. the $y-$check algorithm returns a feasible solution for a candidate. In this case we found a feasible SPP(L) solution, so we terminate BLUE with a proof of optimality;

2. we prove that no feasible solution exists, because we explore the complete tree and the $y-$check algorithm returns infeasible for all candidates. We thus increase L and re-iterate BLUE;

3. we explore a maximum number of nodes, or the $y-$check algorithm fails for a candidate (and does not find a feasible solution for any other attempted candidate). In this case we proceed with the Benders' decomposition described below.

This B&B is effective on the so-called perfect packing instances, i.e., on those instances in which the optimal solution has no waste. Thus, the maximum number of explored nodes is set to 10^7 nodes if the instance appears to be a perfect packing, i.e., if the best lower bound L is equal to $\sum_{j \in \mathcal{N}} w_j h_j / WH$, and to $5 \cdot 10^4$ otherwise. Other attempts that we made to speed up the algorithm, as, e.g., using dual feasible functions at any node of the tree, or selecting the bottom-most niche instead of the left-most column, led to slightly worse computational results.

Benders' Decomposition for the SPP(L).

We solve the MILP (22)–(26) of Section 2.3, by lifting constraints (25) to (33) as described in Section 4. As noticed before, we work on a fixed height L instead than on the minimization of the strip height. This is simply obtained by disregarding the objective function, and setting $z = L$ in (24).

There are two ways to solve MILPs involving exponentially many constraints. In a modern *branch-and-cut* implementation, the violated constraint is added as soon as possible to the model, by using callbacks invoked at the nodes of the MILP enumeration tree. In the standard *delayed cut generation method*, the master is solved to optimality, then one finds violated cuts, if any, adds them to the model and re-optimizes the master, until no further violation exists. We attempted both implementations, and obtained better results with the delayed cut generation method. In our opinion, the reason for the better behavior of the latter method is that it allows the solver to use completely the automated preprocessing techniques, which are very effective for the SPP(L), and the *dynamic search* method, which is much faster than the standard branch-and-cut in the instances that we tested.

Whenever the MILP returns a feasible solution, we check if this is also feasible for SPP(L) by invoking the $y-$check algorithm with a maximum number of $2 \cdot 10^7$ iterations. If the $y-$check algorithm fails for a candidate instance, then we suppose the instance is infeasible, add the corresponding cut(s), and continue the search. Note that we can still provide a proven optimal solution if we find another solution at the same height L which is feasible for $y-$check.

Similarly to the B&B, also the Benders' decomposition has three exits:

1. it finds a feasible, thus optimal, SPP(L) solution, so BLUE terminates;

2. it proves that no solution exists at height L, because, after possibly inserting cuts, the MILP returns infeasible, thus BLUE increases L and re-iterates;

3. it reaches a time limit, or the $y-$check algorithm fails for a candidate (and does not find a feasible solution for any other candidate), so BLUE terminates with a heuristic solution.

We also use two further enhancements. First, we decrease the number of variables as follows:

1. We select the item j having the largest number of variables, and impose it to be packed in the left half of the strip by removing all variables x_{jp} with $p \geq W/2$;

2. Suppose that an item j may be packed in a column p, but the area at the right of j would be lost, because all items have width greater than $W - p - w_j$. Further suppose that there is another column $q > p$ in which j can be packed. In this case we remove variable x_{jp} from the program, because any solution in which j is packed in p may be transformed into an equivalent one in which j is packed in q.

The second enhancement is a MILP-based heuristic, which is invoked before the decomposition for the instances with identical items (i.e., items having same width and height). We consider again the MILP (22)–(26) and solve it as discussed, but replace the binary variables x_{jp} with integer ones. In particular, suppose there are $k(j)$ items identical to item j and having index greater than j, then all these items are removed from the instance, and variables x_{jp} are constrained to be integer such that $0 \leq x_{jp} \leq 1 + k(j)$, for $p \in \mathcal{W}(j)$. After the model is solved, the values of the integer variables are easily mapped back into the original binary ones, and the solution is given to the $y-$check algorithm. If feasible, then the heuristic ends with an optimal SPP(L) solution (and BLUE terminates), otherwise valid cuts are searched with the procedure of Section 4, mapped again on the integer variables, and inserted into the model. The resulting algorithm is just a heuristic because the Benders' cuts may forbid feasible solutions, but sometimes achieves feasible solutions in short time.

6 Computational Results

All algorithms have been implemented in C++ and run on an Intel Core(TM)2 Quad CPU Q8200, running at 2.33 GHz under Linux openSUSE 11.4 operative system. The LPs and the MILPs were solved with IBM Ilog Cplex 12.5, by setting parameters RepeatPresolve = 3, Reduce = 3, Probe = 3, Simmetry = 5, and imposing it to use a single processor (Threads = 1). The subset sum problems were solved using a standard dynamic programming, and the knapsack problems with procedure `combo` by Martello et al. (1999). Algorithm BLUE was allowed a time limit of 1200 CPU seconds on each instance.

6.1 Comparison with existing methods

We tested our algorithm on the benchmark instances that have been addressed with exact methods in the SPP literature, and compare with the previous algorithms. We refer to Boschetti and Montaletti (2010), and references therein, for the details on the original papers that provided the benchmark instances. We address in total 560 instances, so we only provide here a summarized information of our results, but refer to the appendix, and also to our web site `www.or.unimore.it\resources\SPP.html`, for more details.

In Table 1 we give a summary of our results, and compare with the following algorithms:

- MMV03 = Martello et al. (2003),
- BKC07 = the most performing algorithm (DA) by Bekrar et al. (2007),
- APT09 = Alvarez-Valdes et al. (2009),
- KINYN09 = the most performing SPP algorithm (G-STAIRCASE) by Kenmochi et al. (2009),
- BM10 = Boschetti and Montaletti (2010),
- CO11 = the most performing algorithm (DS) by Castro and Oliveira (2011), and
- AIT12 = Arahori et al. (2012, forthcoming).

Under the name of each contribution, we report the speed of the computer that was used, and the maximum time limit in seconds that was allowed. For each benchmark set, we report the name of the set and the number of instances (#). For each algorithm and each set, we report in column "opt" the number of optimal solutions, and in column "sec" the average CPU time in seconds (computed only on the instances optimally solved by that algorithm). The highest number of optimal solutions for each set is reported in bold.

Benchmark sets ngcut, ht and beng are relatively easy. Both BM10, AIT12 and BLUE solve all the instances in these sets to proven optimality in short time, while the remaining approaches fail for a few instances. Our algorithm is almost as fast as AIT12, and much faster than BM10. We refer to the appendix for the detailed results on these "easy" sets. On the other benchmark sets, which are more difficult, BLUE obtains better results than all previous algorithms, and always improves the number of proven optimal solutions within short computational times. Some more details on the results on these sets are presented in the next tables.

Table 1: Summary of the computational results.

		MMV03 0.8 GHz t.l.=3600s		BKC07 1.7 GHz t.l.=1200s		APT09 2 GHz t.l.=1200s		KINYN09 3 GHz t.l.=3600s		BM10 1.6 GHz t.l.=1200s		CO11 2.5 GHz t.l.=3600s		AIT12 (*)		BLUE 2.33 GHz t.l.=1200s	
name	#	opt	sec	opt	sec	opt	sec	opt	sec	opt	sec	opt	sec	opt	sec	opt	sec
ngcut	12	11	118.35	**12**	582.89	**12**	7.33	10	224.19	**12**	35.11	6	21.51	**12**	0.32	**12**	0.20
ht	9	7	514.33	7	288.86	8	1.58	**9**	8.63	**9**	350.00	7	558.77	**9**	2.64	**9**	6.47
beng	10	6	318.86	2	304.21	**10**	1.03	9	4.05	**10**	138.29			**10**	0.97	**10**	0.94
cgcut	3	1	11.48	1	323.54	1	0.00	1	0.12	1	0.69			2	407.86	**3**	11.23
gcut-01-04	4	2	0.00	1	0.00	2	0.00			3	2.47			3	85.19	**4**	2.71
gcut-05-13	9									6	0.76					**7**	28.72
n	13							8	39.20	4	345.04					**10**	7.81
"10 classes"	500					256	21.40			274	34.72			142	112.61	**322**	26.11

(*)=t.l.=3600 sec on a 3.3GHz for ngcut, ht, beng, cgcut, gcut and n; t.l.=1200 sec on a 2.53GHz for the 10 classes.

Table 2 gives the details of the results for the cgcut instances by Christofides and Whitlock (1977). In column "opt" we report a "*" if the instance is solved to proven optimality, in column "sec" we give the computational time, and for BLUE we also report the optimal solution value z. For BKC07 we write *n.a.* when their algorithm converged to a sub-optimal solution (a correct optimal solution for this case was later provided by AIT12). The first instance is easily solved to optimality by all algorithms. The second one was solved for the first time by AIT12 in slightly more than 800 seconds, whereas BLUE only needs about 20 seconds. The third instance was still an open problem, but BLUE solves it in less than 13 seconds.

Table 2: Results and comparison on the cgcut instances.

			MMV03 0.8 GHz t.l.=3600s		BKC07 1.7 GHz t.l.=3600s		APT08 2 GHz t.l.=1200s		KINYN09 3 GHz t.l.=3600s		BM10 1.6 GHz t.l.=1200s		AIT12 3.3 GHz t.l.=3600s		BLUE 2.33 GHz t.l.=1200s		
name	n	W	opt	sec	opt	sec	opt	sec	opt	sec	opt	sec	opt	sec	z	opt	sec
cgcut01	16	10	*	11.48	*	323.54	*	0.00	*	0.12	*	0.69	*	0.00	23	*	0.03
cgcut02	23	70		*t.lim.*		*n.a.*		*t.lim.*		*t.lim.*		*t.lim.*	*	815.71	64	*	20.74
cgcut03	62	70		*t.lim.*		*t.lim.*		*t.lim.*		*t.lim.*		*t.lim.*		*t.lim.*	656	*	12.91

Table 3 gives the details of the results on the gcut instances by Beasley (1985). When BLUE fails in closing the instance, in column z we provide the interval given by the lower and upper bound that it found. The first four instances have been addressed by all the exact algorithms in the table. Among them, instances gcut01 and gcut03 are very easy and were solved by almost all algorithms in a short time. Instance gcut02 was already solved by BM10 in 7 seconds, and by AIT12 in 255 seconds, whereas we need less than 2 seconds. Instance gcut04 was still an open problem, and

we solve it in less than 9 seconds. The remaining nine instances are characterized by a large width, from 500 to 3000. They were addressed just by BM10, that could solve six of them, whereas we can solve one instance more, gcut11, by using a similar computational effort.

Table 3: Results and comparison on the gcut instances.

			MMV03 0.8 GHz t.l.=3600s		BKC07 1.7 GHz t.l.=3600s		APT08 2 GHz t.l.=1200s		BM10 1.6 GHz t.l.=1200s		AIT12 3.3 GHz t.l.=3600s		BLUE 2.33 GHz t.l.=1200s		
name	n	W	opt	sec	opt	sec	opt	sec	opt	sec	opt	sec	z	opt	sec
gcut01	10	250	*	0.00	*	0.00	*	0.00	*	0.00	*	0.01	1016	*	0.01
gcut02	20	250		*t.lim.*		*n.a.*		*t.lim.*	*	7.41	*	255.48	1187	*	1.76
gcut03	30	250	*	0.00		*t.lim.*	*	0.00	*	0.00	*	0.09	1803	*	0.17
gcut04	50	250		*t.lim.*		*t.lim.*		*t.lim.*		*t.lim.*		*t.lim.*	2995	*	8.88
gcut05	10	500							*	0.69			1273	*	0.30
gcut06	20	500							*	1.33			2622	*	1.11
gcut07	30	500							*	0.36			4693	*	0.56
gcut08	50	500								*t.lim.*			[5824, 5904]		*t.lim.*
gcut09	10	1000							*	0.09			2317	*	0.12
gcut10	20	1000							*	2.08			5964	*	35.63
gcut11	30	1000								*t.lim.*			6866	*	163.10
gcut12	50	1000							*	0.00			14690	*	0.19
gcut13	32	3000								*t.lim.*			[4803, 4945]		*t.lim.*

The optimal solutions of the two most important open problems that we closed in these two sets, namely cgcut03 and gcut04, are depicted in the appendix. They are very interesting, because characterized by complex non-guillotine structures, that create large holes and make it difficult to compute both the lower and the upper bound. The solutions that we obtained on all other instances are available on our web site.

The details of the results on the n instances by Burke et al. (2004) are given in Table 4. In terms of exact algorithms, this set was addressed only by KINYN09 (just the first 12 instances, using algorithm StaircasePP) and BM10. Algorithm BLUE largely outperforms these two exact algorithms, solving more instances to proven optimality, using usually less time. Note that these instances have been built ad hoc to have zero waste, and hence represent an interesting test bed more for heuristics than for exact algorithms (indeed, that was their original scope), due to the fact that the computation of sophisticated lower bounds is useless.

The details of the results on the 10 classes proposed by Berkey and Wang (1987) and Martello and Vigo (1998) are given in Table 5. Each class contains 50 instances, divided into 5 groups of 10 instances, one for each value of $n \in \{20, 40, 60, 80, 100\}$. These sets have been addressed by APT08, BM10, and AIT12. Each line in the table gives the number of optimal solutions and average time (for the instances solved to

Table 4: Results and comparison on the n instances.

			KINYN09 3 GHz t.l.=3600s		BM10 1.6 GHz t.l.=1200s		BLUE 2.33 GHz t.l.=1200s		
name	n	W	opt	sec	opt	sec	z	opt	sec
n01	10	40	*	0.27	*	0	40	*	0.01
n02	20	30	*	0.08	*	2.16	50	*	0.04
n03	30	30	*	289.09	*	770.55	50	*	0.11
n04	40	80	*	23.02		*t.lim.*	80	*	0.16
n05	50	100		*t.lim.*		*t.lim.*	100	*	4.63
n06	60	50	*	0.05		*t.lim.*	100	*	4.21
n07	70	80	*	0.11	*	607.45	100	*	16.79
n08	80	100		*t.lim.*		*t.lim.*	[80; 83]		*t.lim.*
n09	100	50		*t.lim.*		*t.lim.*	150	*	38.13
n10	200	70	*	0.22		*t.lim.*	150	*	6.31
n11	300	70	*	0.74		*t.lim.*	150	*	7.71
n12	500	100		*t.lim.*		*t.lim.*	[300; 311]		*t.lim.*
n13	3152	640				*t.lim.*	[960; 988]		*t.lim.*

proven optimality), for each group and each algorithm. We present results only for those groups in which at least one instance was solved by one of the algorithms. BLUE is on average faster and provides a higher number of proven optimal solutions than the previous algorithms. Notably, it solves for the first time all instances with $n = 20$ for the classes 3, 4, and 5.

6.2 Evaluation of the behavior of BLUE

The most effective approaches published in the SPP literature are branch-and-bound algorithms based on the idea of building solutions by packing one item at a time in the strip. Algorithm BLUE has a completely different approach, that, intuitively, divides items into slices, packs slices, and then attempts the reconstruction of the original items. This innovative approach appears to perform better on all benchmark instances. This can be noted, for example, in Figure 4. The figure focuses again on the 10 classes, and graphically depicts the number of optimal solutions for each algorithm and each class, from the most difficult to the easiest. It can be noted that BLUE provides equal or better results than the other algorithms on each class.

It is important to notice that all the components of BLUE contributes to the good results. The root node solves to optimality 279 instances out of 560, with an average time of about 10 seconds. Our new upper bound U_2 improves 154 times the upper bound U_1 that we derived from the literature. The new lower bound L_4 improves 122 times the maximum value among the first three lower bounds that we took from the literature (L_1, L_2, and L_3), and then L_5 obtains other 15 further improvements.

The main loop decreases the upper bound in 49 cases, and, most important,

Table 5: Results and comparison on the 10 classes (10 instances per line).

			APT08 2 GHz t.l.=1200s		BM10 1.6 GHz t.l.=1200s		AIT12 2.53 GHz t.l.=1200s		BLUE 2.33 GHz t.l.=1200s	
class	n	W	opt	sec	opt	sec	opt	sec	opt	sec
1	20	10	10	2.75	10	0.67	10	137.36	10	0.44
	40	10	10	6.29	10	19.55	4	0.48	10	0.12
	60	10	7	28.71	10	39.97	2	4.97	10	0.25
	80	10	9	105.30	10	14.26	1	0.09	10	0.44
	100	10	5	93.14	10	95.68	2	0.04	10	0.92
2	20	30	9	0.40	10	75.75	10	1.69	10	2.68
	40	30	9	0.48	10	103.55	10	0.01	10	0.13
	60	30	8	4.68	10	146.93	10	0.02	10	0.32
	80	30	8	2.63	10	101.84	10	0.01	10	0.42
	100	30	9	5.32	10	81.93	10	0.04	10	0.51
3	20	40	8	229.49	9	77.01	9	292.73	10	15.13
	40	40	6	0.44	6	0.37	4	12.17	9	113.45
	60	40	4	2.38	4	19.23	0		9	72.18
	80	40	5	1.79	5	12.97	0		8	2.64
	100	40	6	2.84	6	20.99	0		7	4.68
4	20	100	1	8.00	1	608.88	8	371.77	10	171.68
	40	100	0		0		1	528.75	0	
5	20	100	8	148.75	6	10.06	7	162.17	10	66.92
	40	100	7	1.45	8	12.50	7	48.36	10	4.18
	60	100	6	1.39	7	61.03	1	362.40	8	1.30
	80	100	6	1.83	6	10.23	1	0.10	8	1.57
	100	100	4	11.32	5	41.25	0		7	21.15
6	20	300	0		0		1	755.45	5	196.91
7	20	100	10	0.42	10	0.08	10	59.39	10	0.08
	40	100	10	0.59	10	0.42	1	325.52	10	0.20
	60	100	10	1.33	10	0.44	0		10	0.45
	80	100	10	1.03	10	0.79	0		10	1.29
	100	100	10	3.23	10	2.07	0		10	1.39
8	20	100	1	10.55	1	37.67	2	303.65	2	6.66
9	20	100	10	0.01	10	0.00	10	27.16	10	0.04
	40	100	10	0.54	10	0.08	2	258.60	10	0.20
	60	100	10	0.02	10	0.00	0		10	0.25
	80	100	10	0.14	10	0.11	0		10	0.37
	100	100	10	0.16	10	0.12	0		10	0.60
10	20	100	7	56.42	7	20.28	8	422.23	9	75.17
	40	100	2	16.46	2	1.48	1	115.44	6	247.68
	60	100	1	0.44	1	62.33	0		3	69.84
	80	100	0		0		0		1	453.11

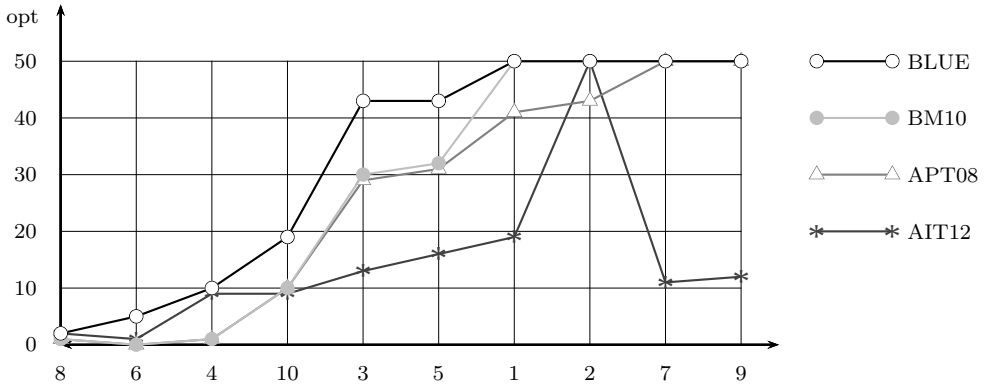

Figure 4: Number of optimal solutions (out of 50) per class, from the most difficult to the easiest.

increases the lower bound in 106 cases. In this way it solves to optimality other 98 instances. The loop is iterated on average just once per instance. Inside the loop, the feasibility or infeasibility of an instance is proven 39 times by the branch-and-bound, and 328 times by the Benders' decomposition. Overall, the Benders' decomposition is the most important part of the algorithm, because it performs very well on the difficult instances.

The two enhancements that we developed to speed up this decomposition are both important. The strategies to decrease the number of variables are effective on instances having large W. For example, for gcut04 they reduce the number of variables from 1640 to just 424. The MILP-based heuristic is also successful in a few important cases. For example, it finds the optimal solution of cgcut03 in just a few seconds.

In Figure 5 and Table 6 we show some insight about two of the most innovative components of BLUE, namely, the $y-$check algorithm and the procedure to lift the Benders' cuts. Considering the overall run on the 560 instances, the $y-$check algorithm has been invoked 2490 times by the exact algorithms of Section 5.2. It terminated within the given iteration limit for 96.6% of the cases. In Figure 5 we show the evolution, in terms of percentage of instances solved, for the first 2 seconds of computation. In just 0.01 seconds the algorithm solves 40% of the instances. This is caused by the fact that the algorithm is very quick in finding a feasible solution, if any: It takes just 0.01 seconds on average and 0.24 in the worst case to prove feasibility. Then the task becomes harder, but still the algorithm needs just 0.8 seconds on average to prove the infeasibility of an instance. Being the problem strongly $\mathcal{NP}$-hard, it is not surprising that a few $y-$check instances remain unsolved after one minute.

In Table 6, the impact of the procedures to combine and lift the Bender's cuts is

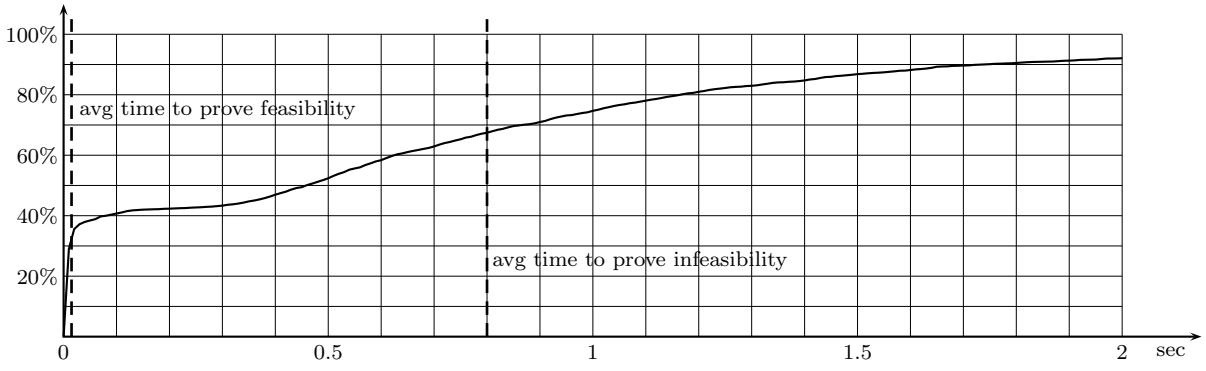

Figure 5: Evolution of the $y-$check algorithm in the first 2 seconds of computation.

evaluated on a few successful examples, taken from instances in the 10 classes that were not solved to proven optimality by previously published algorithms. We compare the results obtained by a version of BLUE that uses the Benders' cuts (20), another version that uses the combinatorial Benders' (21) obtained by the procedure of Section 4.1, and the final version that uses the lifted combinatorial Benders' (33). For each case and each instance we report the optimality of the solution (opt), the computational time (sec), the total time spent inside the MILP of the decomposition (sec MILP), the number of added cuts (num cuts), the number of calls to the $y-$check algorithm (num y-ch.) and the total seconds it elapsed (sec y-ch.).

The standard cuts solve 3 out of 6 instances. The combinatorial cuts do not increase the number of optimal solutions, but can decreases consistently the computational effort, as happens for example in instance 05-40-01, where the time decreases from 610 to 17 seconds. The lifted cuts improve consistently the previous configurations, solving all instances to optimality and decreasing the computational effort. The cuts are efficient in both strengthening the formulation, and thus reducing sec MILP, and in reducing the number of calls and the time spent for problem $y-$check.

7 Conclusions

We proposed an innovative algorithm for the exact solution of the Strip Packing Problem, which is based on a Benders' decomposition and is enriched with several tailored techniques. We proved that the slave problem arising in the decomposition is difficult, but solved it with an algorithm that is efficient in practice. We improved the standard Benders' cuts by using the concept of combinatorial Benders' cuts, and a new lifting procedure which is very effective on the difficult instances.

Table 6: Evaluation of the impact of the different cuts on some previously unsolved instances from the 10 classes.

	Benders' cuts (20)						Combinatorial Benders' (21)						Lifted combinatorial Benders' (33)					
name	opt	sec	sec MILP	num cuts	num y-ch.	sec y-ch.	opt	sec	sec MILP	num cuts	num y-ch.	sec y-ch.	opt	sec	sec MILP	num cuts	num y-ch.	sec y-ch.
05-40-01	*	610.65	607.70	1259	1260	1.70	*	17.32	12.56	494	39	0.01	*	11.80	5.17	400	17	0.23
06-20-05		*t.lim.*	1192.45	1124	1127	0.26		*t.lim.*	590.58	6786	525	0.09	*	146.92	138.24	600	28	0.00
06-20-08	*	662.63	645.97	8	9	0.00	*	657.94	640.99	52	5	0.00	*	604.64	586.98	25	2	0.00
10-40-08		*t.lim.*	564.82	857	857	631.91		*t.lim.*	125.83	2340	181	152.21	*	551.48	177.05	1402	59	129.47
10-40-10	*	169.38	167.89	185	186	0.12	*	240.64	147.6	2197	170	0.18	*	110.81	81.88	1625	66	0.09
10-60-08		*t.lim.*	1145.65	742	742	50.40		*t.lim.*	101.85	316	71	2.86	*	208.67	44.36	216	13	4.17

The proposed algorithm consistently outperforms the previously published approaches. It provides a larger number of optimal solutions in similar or smaller computational effort, and solves for the first time to proven optimality instances that were open since decades.

The general framework that we propose can be adapted to solve other two-dimensional packing problem, such as the two-dimensional knapsack and the two-dimensional bin packing. It can be also generalized to higher dimensional problems. These represent interesting future research directions.

References

Alvarez-Valdes, R., F. Parreño, J. Tamarit. 2009. A branch and bound algorithm for the strip packing problem. *OR Spectrum* **31** 431–459.

Arahori, Y., T. Imamichi, H. Nagamochi. 2012, forthcoming. An exact strip packing algorithm based on canonical forms. *Computers & Operations Research* doi: 10.1016/j.cor.2012.03.003.

Augustine, J., S. Banerjee, S. Irani. 2009. Strip packing with precedence constraints and strip packing with release times. *Theoretical Computer Science* **410** 3792–3803.

Beasley, J. E. 1985. Algorithms for unconstrained two-dimensional guillotine cutting. *Journal of the Operational Research Society* **36** 297–306.

Bekrar, A., I. Kacem, C. Chu. 2007. A comparative study of exact algorithms for the two dimensional strip packing problem. *Journal of Industrial and Systems Engineering* **1** 151 – 170.

Benders, J.F. 1962. Partitioning procedures for solving mixed variables programming problems. *Numerische Mathematik* **4** 238–252.

Berkey, J. O., P. Y. Wang. 1987. Two dimensional finite bin packing algorithms. *Journal of the Operational Research Society* **38** 423–429.

Boschetti, M.A., L. Montaletti. 2010. An exact algorithm for the two-dimensional strip-packing problem. *Operations Research* **58** 1774–1791.

Burke, E.K., M.R. Hyde, G. Kendall. 2011. A squeaky wheel optimisation methodology for two-dimensional strip packing. *Computers & Operations Research* **38** 1035 – 1044.

Burke, E.K., G. Kendall, G. Whitwell. 2004. A new placement heuristic for the orthogonal stock-cutting problem. *Operations Research* **52** 655–671.

Caprara, A., M. Monaci. 2009. Bidimensional packing by bilinear programming. *Mathematical Programming* **118** 75–108.

Castro, P.M., J.F. Oliveira. 2011. Scheduling inspired models for two-dimensional packing problems. *European Journal of Operational Research* **215** 45 – 56.

Christofides, N., C. Whitlock. 1977. An algorithm for two-dimensional cutting problems. *Operations Research* **25** 30–44.

Clautiaux, F., J. Carlier, A. Moukrim. 2007. A new exact method for the two-dimensional orthogonal packing problem. *European Journal of Operational Research* **183** 1196 – 1211.

Codato, G., M. Fischetti. 2006. Combinatorial benders' cuts for mixed-integer linear programming. *Operations Research* **54** 756–766.

Geoffrion, A.M. 1972. Generalized benders decomposition. *Journal of Optimization Theory and Applications* **10**(4) 237–260.

Gilmore, P.C., R.E. Gomory. 1961. A linear programming approach to the cutting-stock problem. *Operations Research* **9** 849–859.

Gilmore, P.C., R.E. Gomory. 1965. Multistage cutting stock problems of two and more dimensions. *Operations Research* **13** 94–120.

Grandcolas, S., C. Pinto. 2010. A SAT encoding for multi-dimensional packing problems. A. Lodi, M. Milano, P. Toth, eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, *Lecture Notes in Computer Science*, vol. 6140. Springer, 141–146.

Harren, R., K. Jansen, L. Prädel, R. Van Stee. 2011. A $(5/3 + \varepsilon)$-approximation for strip packing. *Proceedings of the 12th international conference on Algorithms and data structures*. WADS'11, Springer-Verlag, Berlin, Heidelberg, 475–487.

Herz, J.C. 1972. Recursive computational procedure for two-dimensional stock cutting. *IBM Journal of Research and Development* **16** 462–469.

Hooker, J.N. 2000. *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*. J. Wiley & Sons.

Hooker, J.N. 2007. Planning and scheduling by logic-based benders decomposition. *Operations Research* **55** 588–602.

Hooker, J.N., G. Ottosson. 2003. Logic-based benders decomposition. *Mathematical Programming* **96** 33–60.

Iori, M., J.J. Salazar González, D. Vigo. 2007. An exact approach for the vehicle routing problem with two-dimensional loading constraints. *Transportation Science* **41** 253–264.

Jain, V., I.E. Grossmann. 2001. Algorithms for hybrid MILP/CP models for a class of optimization problems. *INFORMS Journal on Computing* **13** 258–276.

Jansen, K., R. Solis-Oba. 2009. Rectangle packing with one-dimensional resource augmentation. *Discrete Optimization* **6** 310 – 323.

Kenmochi, M., T. Imamichi, K. Nonobe, M. Yagiura, H. Nagamochi. 2009. Exact algorithms for the two-dimensional strip packing problem with and without rotations. *European Journal of Operational Research* **198** 73 – 83.

Kenyon, C., E. Rémila. 2000. A near-optimal solution to a two-dimensional cutting stock problem. *Mathematics of Operations Research* **25** 645–656.

Lesh, N., J. Marks, A. McMahon, M. Mitzenmacher. 2004. Exhaustive approaches to 2d rectangular perfect packings. *Information Processing Letters* **90** 7 – 14.

Leung, S.C.H., D. Zhang, K.M. Sim. 2011. A two-stage intelligent search algorithm for the two-dimensional strip packing problem. *European Journal of Operational Research* **215** 57 – 69.

Martello, S., M. Monaci, D. Vigo. 2003. An exact approach to the strip packing problem. *INFORMS Journal on Computing* **15** 310–319.

Martello, S., D. Pisinger, P. Toth. 1999. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science* **45** 414–424.

Martello, S., D. Vigo. 1998. Exact solution of the two-dimensional finite bin packing problem. *Management Science* **44** 388–399.

Mesyagutov, M.A., E.A. Mukhacheva, G.N. Belov, Scheithauer G. 2011. Packing of one-dimensional bins with contiguous selection of identical items: An exact method of optimal solution. *Automation and Remote Control* **72** 141–159.

Sawaya, N.W., I.E. Grossmann. 2005. A cutting plane method for solving linear generalized disjunctive programming problems. *Computers & Chemical Engineering* **29** 1891 – 1913.

Soh, T., K. Inoue, N. Tamura, M. Banbara, H. Nabeshima. 2010. A sat-based method for solving the two-dimensional strip packing problem. *Fundamenta Informaticae* **102** 467 – 487.

Wei, L., W.-C. Oon, W. Zhu, A. Lim. 2011. A skyline heuristic for the 2d rectangular packing and strip packing problems. *European Journal of Operational Research* **215** 337 – 346.

Westerlund, J., L.G. Papageorgiou, T. Westerlund. 2007. A MILP model for N-dimensional allocation. *Computers & Chemical Engineering* **31** 1702 – 1714.

8 Appendix

This electronic companion is structured as follows. In Section 8.1, we give the proof of Theorem 1. In Section 8.2 we graphically depict two optimal solutions whose structure was commented in the paper. In Section 8.3 we provide additional details on the computational results that we obtained.

8.1 Proof of Theorem 1

In order to prove that $y-$check is strongly $\mathcal{N}P-$complete, we give a polynomial transformation from the following problem.

Definition 5 3-PARTITION: *Given $\overline{n} = 3m$ items, each having weight $s_j \in Z^+$ $(j = 1, 2, \ldots, \overline{n})$, and a value $B \in Z^+$ such that $\sum_{j=1}^{\overline{n}} s_j = 3B$, find a partition of the items into m disjoint subsets $S_1, S_2, \ldots, S_m$ such that $\sum_{j \in S_i} s_j = B$ holds for $i = 1, 2, \ldots, m$, if any.*

In the proof of Lemma 1 we used nine items to obtain two $1 \times B$ buckets. Here we want to obtain m $1 \times B$ buckets, using an iterative method that adds nine more items at a time. We start by considering two $5 \times (2B + 3)$ rectangles, each obtained by packing nine items as those of Figure 2 (see the 18 hatched items in Figure 6), and we embed them into nine new items following the same scheme used in Lemma 1. In this scheme, however, the new items produce two buckets of width five. In Figure 6 we depict this frame, by drawing in white the new items.

In details, let us call $1', 2', \ldots, 9'$ and $1'', 2'', \ldots, 9''$ the hatched items in the two buckets. Items $1'$ and $1''$ cannot be packed in the same $5 \times (2B + 3)$ bucket, because they are too high. As a consequence also items $3'$ and $3''$ must be packed in different buckets. The same reasoning applies to items $2'$ and $2''$, and consequently to items $5'$ and $5''$. Continuing this reasoning one can show that also the remaining items must be packed as in Figure 2. We have thus created two copies of the packing of Figure 2 and four $1 \times B$ empty buckets. We now consider the resulting solution as a single $9 \times (4B + 9)$ rectangle and we embed two of them in a frame of other nine items. We continue this process for, say, k times, until we create 2^k $1 \times B$ empty buckets with $2^k \geq m$ (the technical details on the widths, heights and $x-$coordinates of the items are given at the end of this proof).

Let α denote the number of items used to create the 2^k $1 \times B$ buckets. We complete the instance by adding $\overline{n}$ items with $w_j = 1$, $h_j = s_{j-\alpha}$ $(j = \alpha + 1, \alpha + 2, \ldots, \alpha + \overline{n})$ and p_j^s corresponding to the $x-$coordinate of the empty buckets, and other $2^k - m$ items with $w_j = 1$, $h_j = B$ and the same p_j^s of the previous ones. Each of the last $2^k - m$ items completely fills $2^k - m$ buckets, so leaving exactly m empty buckets.

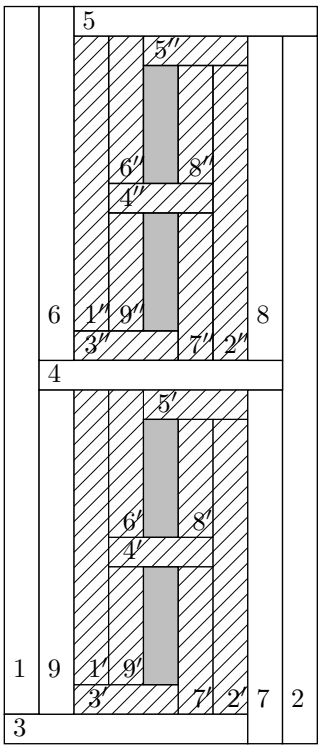

Figure 6: Basic frame for the reduction of *3-Partition* to $y-$check.

These can be feasibly filled by the remaining items if and only if 3-Partition has a feasible solution. Since 3-Partition is strongly $\mathcal{N}P-$complete, the same holds for $y-$check.

Details on sizes and coordinates of the items.

We constructed the instance by adding for k times nine items as those depicted in Figure 2. Let (i, j) denote the item of "type" i ($i = 1, 2, \ldots, 9$) used at iteration j ($j = 1, 2, \ldots, k$). The items of type 1, 2, 6, 7, and 8 have width one, but height depending on the iteration, the items of type 3, 4, and 5 have height one, but width

depending on the iteration. In particular:

$$\begin{array}{lll}
w_{(i,j)} = 1,\ h_{(i,j)} = \ 2^j B + 3(2^j - 1) - 1 & & i = 1, 2 \\
w_{(i,j)} = 1,\ h_{(i,j)} = h_{(1,j)}/2 & & i = 6, 7 \\
w_{(i,j)} = 1,\ h_{(i,j)} = h_{(6,j)} - 1 & & i = 8, 9 \\
h_{(i,j)} = 1,\ w_{(i,j)} = \left\{ \begin{array}{ll} 3 & \text{for } j = 1 \\ 3 + 2^j & \text{for } j > 1 \end{array} \right. & & i = 3, 4, 5.
\end{array}$$

The $x-$coordinate of an item (i, j) is:

$$\begin{array}{ll}
2(k - j) & i = 1, 3 \\
1 + 2(k - j) & i = 4, 6, 9 \\
2 + 2(k - j) & i = 5 \\
2^k - 2(k - j) & i = 2 \\
2^k - 1 - 2(k - j) & i = 7, 8.
\end{array}$$

This information concludes the proof. □

8.2 Graphical representation of two optimal solutions

In Figure 7 we depict an optimal solution of instance cgcut03 by Christofides and Whitlock (1977), and in Figure 8 an optimal solution of instance gcut04 by Beasley (1985). Both figures are scaled, so that a unit on the height is one half of a unit on the width.

As described in the paper, these two solutions are characterized by complex non-guillotine structures, that create large holes and make difficult the computation of both the lower and the upper bound. The solutions that we obtained on all other instances, either proven optimal or heuristic, are available for download on our web site `www.or.unimore.it\resources\SPP.html`.

8.3 Additional Computational Results

In Table 7 we give the results of our algorithm on the "easy" benchmark sets ngcut, ht and beng. As done in the paper, in column "opt" we report a "*" if the instance is solved to proven optimality, in column "sec" we give the computational time, and in column z we report the solution value found by BLUE. We recall that we compare with: MMV03 = Martello et al. (2003), BKC07 = the most performing algorithm (DA) by Bekrar et al. (2007), APT09 = Alvarez-Valdes et al. (2009), KINYN09 = the most performing algorithm (G-STAIRCASE) by Kenmochi et al. (2009), BM10 = Boschetti and Montaletti (2010), CO11 = the most performing algorithm (DS) by Castro and Oliveira (2011), and AIT12 = Arahori et al. (2012, forthcoming).

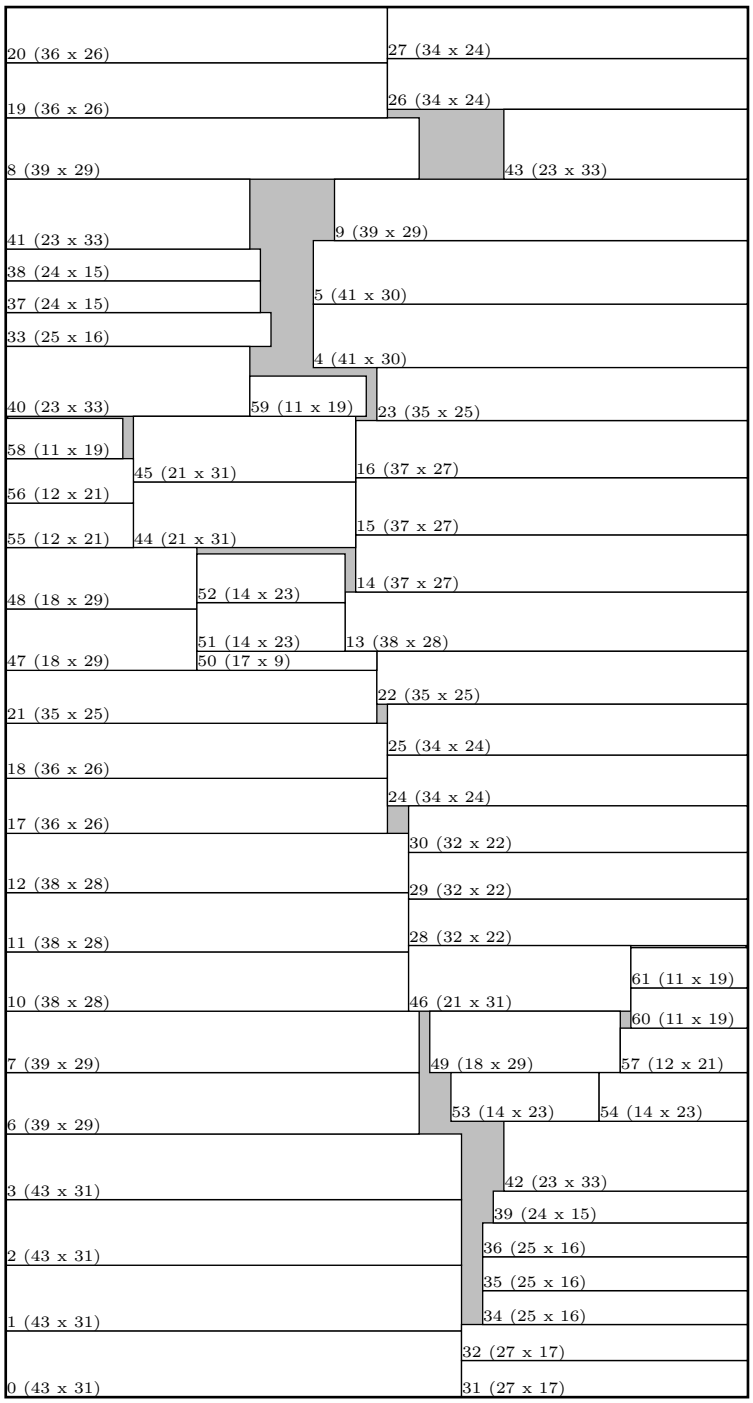

Figure 7: Optimal solution of instance cgcut03 (W=70, $z = 656$).

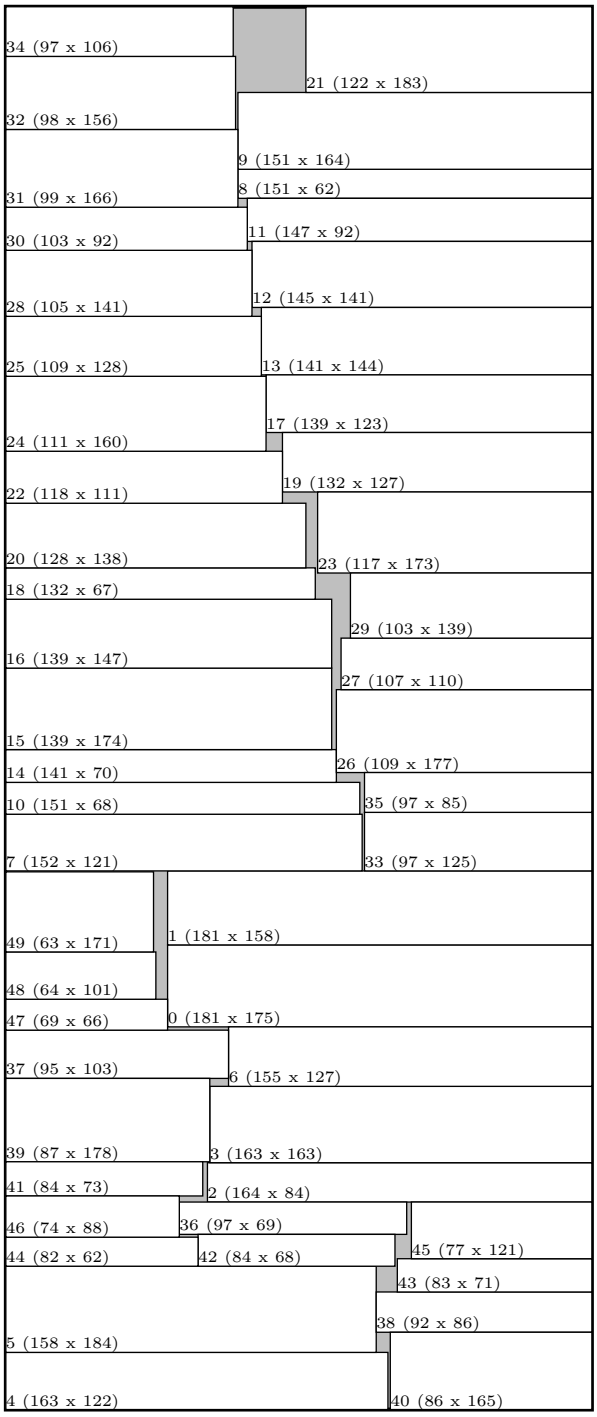

Figure 8: Optimal solution of instance gcut04 (W=250, $z = 2995$).

Table 7: Results and comparison on nguct, ht, and beng instances.

			MMV03 0.8GHz t.l.=3600s		BKC07 1.7GHz t.l.=3600s		APT09 2GHz t.l.=1200s		KINYN09 3GHz t.l.=3600s		BM10 1.6GHz t.l.=1200s		CO11 2.5GHz t.l.=3600s		AIT12 3.3GHz t.l.=3600s		BLUE 2.33GHz t.l.=1200s		
name	n	W	opt	sec	opt	sec	opt	sec	opt	sec	opt	sec	opt	sec	opt	sec	z	opt	sec
ngcut01	10	10	*	0.05	*	23.20	*	2.20	*	0.39	*	0.08	*	81.60	*	0.00	23	*	0.19
ngcut02	17	10	*	11.31	*	1052.72	*	3.10		*t.lim.*	*	0.47		*t.lim.*	*	0.07	30	*	0.08
ngcut03	21	10	*	27.01	*	519.70	*	0.00	*	0.10	*	1.62	*	12.60	*	0.00	28	*	0.03
ngcut04	7	10	*	0.00	*	0.02	*	0.00	*	0.14	*	0.14	*	1.22	*	0.00	20	*	0.04
ngcut05	14	10	*	0.00	*	119.58	*	0.00	*	0.07	*	0.34	*	3.18	*	0.00	36	*	0.02
ngcut06	15	10	*	727.20	*	1079.26	*	4.60	*	147.31	*	0.84		*t.lim.*	*	0.16	31	*	0.41
ngcut07	8	20	*	0.00	*	0.00	*	0.00	*	0.10	*	0.38	*	0.98	*	0.00	20	*	0.01
ngcut08	13	20	*	53.09	*	178.06	*	3.50	*	0.50	*	15.39	*	29.50	*	0.06	33	*	0.36
ngcut09	18	20		*t.lim.*	*	1269.83	*	58.10	*	1971.64	*	286.55		*t.lim.*	*	3.48	50	*	0.67
ngcut10	13	30	*	0.18	*	1152.83	*	2.60	*	113.98	*	6.58		*t.lim.*	*	0.01	80	*	0.06
ngcut11	15	30	*	483.01	*	733.18	*	13.80	*	7.71	*	107.47		*t.lim.*	*	0.04	52	*	0.44
ngcut12	22	30	*	0.00	*	866.32	*	0.00		*t.lim.*	*	1.41		*t.lim.*	*	0.00	87	*	0.03
tot opt/avg sec			11	118.35	**12**	582.89	**12**	7.33	10	224.19	**12**	35.11	6	21.51	**12**	0.32		**12**	0.20
ht01	16	20	*	10.84	*	0.00	*	0.00	*	0.07	*	3.84	*	2.08	*	0.00	20	*	0.02
ht02	17	20	*	3043.25	*	378.81	*	0.40	*	0.07	*	149.98	*	5.28	*	0.00	20	*	0.25
ht03	16	20	*	500.75	*	197.53	*	0.10	*	0.10	*	1.22	*	2.51	*	0.00	20	*	0.03
ht04	25	40	*	8.26	*	874.05	*	0.10	*	0.11	*	611.70	*	91.80	*	0.00	15	*	0.06
ht05	25	40	*	20.29	*	571.65	*	0.10	*	0.06	*	300.95	*	20.40	*	0.00	15	*	0.06
ht06	25	40	*	16.94	*	0.00	*	1.40	*	0.06	*	25.79	*	18.30	*	0.00	15	*	0.05
ht07	28	60		*t.lim.*		*n.a.*	*	1.80	*	0.10	*	654.56	*	3771.00	*	0.01	30	*	0.06
ht08	29	60		*t.lim.*		*n.a.*		*t.lim.*	*	76.97	*	732.05		*t.lim.*	*	23.77	30	*	57.66
ht09	28	60	*	0.00	*	0.00	*	8.70	*	0.13	*	669.90		*t.lim.*	*	0.00	30	*	0.07
tot opt/avg sec			7	514.33	7	288.86	8	1.58	**9**	8.63	**9**	350.00	7	558.77	**9**	2.64		**9**	6.47
beng01	20	25	*	911.37	*	608.41	*	5.50	*	0.93	*	26.95			*	0.25	30	*	0.67
beng02	40	25		*t.lim.*		*t.lim.*	*	0.40	*	22.89	*	72.55			*	5.43	57	*	0.57
beng03	60	25		*t.lim.*		*t.lim.*	*	0.50	*	0.32	*	69.42			*	0.02	84	*	0.21
beng04	80	25		*t.lim.*		*t.lim.*	*	3.30		*t.lim.*	*	198.22			*	0.01	107	*	1.46
beng05	100	25	*	500.62		*t.lim.*	*	0.10	*	0.31	*	73.42			*	0.03	134	*	0.46
beng06	40	40		*t.lim.*		*t.lim.*	*	0.10	*	0.29	*	82.41			*	0.00	36	*	0.18
beng07	80	40	*	0.56		*t.lim.*	*	0.10	*	0.18	*	607.33			*	0.04	67	*	2.47
beng08	120	40	*	500.54		*t.lim.*	*	0.10	*	2.67	*	93.73			*	0.01	101	*	0.72
beng09	160	40	*	0.03	*	0.00	*	0.10	*	2.38	*	76.03			*	3.65	126	*	1.04
beng10	200	40	*	0.03		*n.a.*	*	0.10	*	6.52	*	82.85			*	0.25	156	*	1.60
tot opt/avg sec			6	318.86	2	304.21	**10**	1.03	9	4.05	**10**	138.29			**10**	0.97		**10**	0.94