A Branch-and-Price Algorithm for the Temporal Bin Packing Problem

Mauro Dell'Amico

DISMI, University of Modena and Reggio Emilia, Via Amendola 2, 42122 Reggio Emilia, Italy
mauro.dellamico@unimore.it

Fabio Furini [1]

Université Paris Dauphine, PSL Research University, LAMSADE, 75016 Paris, France
fabio.furini@dauphine.fr

Manuel Iori

DISMI, University of Modena and Reggio Emilia, Via Amendola 2, 42122 Reggio Emilia, Italy
manuel.iori@unimore.it

Abstract

We study an extension of the classical Bin Packing Problem, where each item consumes the bin capacity during a given time window that depends on the item itself. The problem asks for finding the minimum number of bins to pack all the items while respecting the bin capacity at any time instant. A polynomial-size formulation, an exponential-size formulation, and a number of lower and upper bounds are studied. A branch-and-price algorithm for solving the exponential-size formulation is introduced. An overall algorithm combining the different methods is then proposed and tested trough extensive computational experiments.

Keywords: Bin Packing Problem, Branch-and-Price Algorithm, Temporal Bin Packing Problem

1. Introduction

The *Bin Packing Problem* (BPP) is one of the classical problems in combinatorial optimization and has been extensively studied in the literature, see, e.g., Delorme et al. [21]. Given a large number of identical bins of capacity $W \in \mathbb{Z}_+$ and a set $N = \{1, 2, \ldots, n\}$ of items, where each item $j \in N$ is associated with an integer weight $w_j \leq W$, the BPP asks to pack all the items into the minimum number of bins without exceeding the capacity.

In this paper, we study a natural generalization of the BPP called the *Temporal Bin Packing Problem* (TBPP). In the TBPP, a feasible assignment of the items to the bins must be computed over a

[1]Corresponding author

discretized time horizon $\widehat{T} = \{0, 1, 2, \ldots, |\widehat{T}|\}$, of total length $|\widehat{T}|$. If a bin is selected, its capacity is a renewable resource that is available at any time unit in the horizon. Each item $j \in N$ consumes the bin capacity during a given time window $[s_j, t_j)$, with $0 \leq s_j < t_j < |\widehat{T}|$. The integer input parameters $s_j \in \mathbb{Z}_+$ and $t_j \in \mathbb{Z}_+$ represent the *starting time* and the *ending time* of an item, respectively. As for the BPP, in the TBPP each item must be assigned to a unique bin where it remains for its entire time window. The TBPP asks to pack all the items into the minimum number of bins so that the bin capacity is never exceeded at any unit of time. The problem is *strongly* NP-hard, because its restriction obtained by setting $|\widehat{T}| = 1$ boils down to a BPP, which is well known to be strongly NP-hard.

The TBPP adds a temporal dimension to the classical BPP, thus making the problem very challenging to solve in practice. A related difficult problem is the *Vector Packing Problem* (VPP), see, e.g., Hessler et al. [30], where each bin has k capacities $W_1, \ldots, W_k$ and each item j is associated with a vector of k weights $w_j = (w_{j1}, \ldots, w_{jk})$. A feasible VPP solution consists in packing all items in the bins so that the capacity is respected for all the k dimensions. We will show in Section 3 that the TBPP is a special case of the VPP.

Other related problems are: i) the *Two-Dimensional* BPP (2D-BPP), see, e.g. Pisinger and Sigurd [37], in which both items and bins are rectangles and the aim is to pack all items without overlapping in the minimum number of bins; and ii) the *Temporal Knapsack Problem* (TKP), see, e.g., Caprara et al. [11], in which items also have a profit and the aim is to find a subset of items of maximum profit that fits into a single bin. These problems, as well as other interesting related problems, are discussed in more detail in Section 2.

The TBPP finds applications in many fields, including logistics, healthcare, production and warehouse management. Consider, for example, the production field: each item can be interpreted as a task (or a product) that must attain a given production rate (equal to its weight) in each time unit of a given time window; each bin can be seen as a production plant that can be used for the allocation of tasks; minimizing the number of bins consequently implies minimizing the number of production plants that are used for the tasks (see, e.g., Angelelli et al. [4] for a related problem).

The remainder of the paper is organized as follows. Section 2 describes the related literature. Section 3 presents two TBPP mathematical models, the former having polynomial size and the latter exponential. Sections 4 and 5 provide, respectively, lower and upper bounds for the problem, whereas Section 6 describes a branch-and-price algorithm that solves the exponential-size model. All algorithms and models are computationally tested in Section 7 and conclusions are drawn in Section 8.

2. Literature and related problems

The BPP is one of the most widely studied problems in the combinatorial optimization field. A number of surveys and annotated bibliographies have been consequently proposed during the years to describe the main techniques that have been developed for its solution. Such techniques are either focused on the BPP or on its reformulation known as the *Cutting Stock Problem* (CSP), where all items having same weight are grouped together into item types.

Useful classifications have been provided by Wäscher et al. [42], who presented a typology of cutting and packing problems based on detailed categorization criteria, and Coffman Jr. and Csirik [15], who introduced a four-field classification scheme aimed at highlighting the main theoretical results in the area. A few years later, Coffman Jr. et al. [16] presented an overview of approximation algorithms for the BPP and a number of its variants, and classified all references according to Coffman Jr. and Csirik [15]. Valério de Carvalho [39] presented a survey with a focus on the most popular *Linear Programming* (LP) methods for the BPP and the CSP.

Recently, Delorme et al. [21] reviewed the most important mathematical models and algorithms developed for the exact solution of the BPP and the CSP, and experimentally evaluated the performance of the main available software tools. The extensive results obtained, together with the input benchmark instances addressed, have been gathered together and made available on-line at the *Bin Packing Problem Library*, as shown in Delorme et al. [23]. Exact algorithms that appeared after Delorme et al. [23] are the iterative aggregation and disaggregation method by Clautiaux et al. [14], the improved *reflect* formulation of Delorme and Iori [20], and the branch-and-price by Wei et al. [43].

A number of problem extensions have been proposed during the years. We believe it is worth describing the main results that have been obtained on those variants that are close to the TBPP.

The previously mentioned VPP has been the object of several interesting researches. Caprara and Toth [10] focused on the case with $k = 2$ dimensions, providing effective heuristics and a few exact algorithms, the most effective one based on column generation. Alves et al. [2] implemented several dual-feasible functions and fast lower bounding techniques. Brandão and Pedroso [9] used pseudo-polynomial arc-flow models and managed to reduce their size through the use of graph reduction techniques. Very recently, Hessler et al. [30] proposed efficient stabilized branch-and-price algorithms. Their column-generation sub-problem is a multidimensional knapsack problem (see, e.g., Dell'Amico et al. [19]) either binary, bounded, or unbounded, that they solved as a shortest path problem with resource constraints.

Extensions in which items and bins are boxes in d dimensions have also been intensively studied. Most of the works on these problems focused on the case where $d = 2$, solving the 2D-BPP. The aim of the 2D-BPP is to pack all items into the minimum numbers of bins without overlapping. A review of some of the methods to solve the BPP and the 2D-BPP was given in the early nineties by Haessler and Sweeney [29]. Later on, surveys on the 2D-BPP and on some of its relevant variants were proposed by Lodi et al. [32, 33, 34]. Recent relevant results on the 2D-BPP have been obtained, among others, by Pisinger and Sigurd [37], who developed an efficient exact algorithm based on column generation and constraint programming, and by Serairi and Haouari [38], who proposed a list of lower bounding techniques.

The problem of *interval scheduling with a resource constraint* (ISRC) was presented by Angelelli and Filippi [3]. The ISRC is a scheduling problem where jobs have to be processed by parallel identical machines. Similarly to the items in TBPP, each job in the ISRC has fixed start and finish time, as well as a resource consumption (i.e., a weight). Angelelli and Filippi [3] focused on the recognition version of the ISRC, and proved that deciding whether an instance has a feasible solution is strongly NP-complete even when the resource capacity of the machines is fixed to any value greater than or equal to two. A few years later, Angelelli et al. [4] studied the optimization

version of the ISRC, whose objective is a weighted function that depends on the assignment of jobs to machines. They proposed a column generation scheme, as well as greedy and restricted enumeration heuristics, and extensively tested them on a number of instances.

Another relevant problem is the *BPP with Contiguity Constraints* (BPPC). In the BPPC, a certain number of copies might exist for an item, and all copies should be packed in consecutive bins. Starting from Martello et al. [36], the BPPC has been used as a relaxation for two-dimensional cutting and packing problems, either within branch-and-bound algorithms (see, e.g., Alvarez-Valdes et al. [1] and Belov and Rohling [7]) or in combinatorial Benders decompositions (see, e.g., Côté et al. [17] and Delorme et al. [22]).

It is well-known that the BPP can be solved by a *Dantzig-Wolfe* reformulation in which each subproblem is a one-dimensional *Knapsack Problem* (Gilmore and Gomory [26, 27]). The same result holds for the TBPP, with the relevant difference that the subproblem is a TKP. The TKP has received a fair amount of attention in the recent combinatorial optimization literature. The problem was formally introduced in Bartlett et al. [6] to model resource allocation problems in the context of sparse resources, such as communication bandwidth of computer memory. Caprara et al. [11] were the first to solve the TKP with a Dantzig-Wolfe reformulation, using two variants of a branch-and-price algorithm in which subproblems are either associated with groups of capacity constraints or with single capacity constraints, and showing that the former variant performs much better than the latter. Gschwind and Irnich [28] provided improved computational results by producing stabilized column generation algorithms based on the use of dual-optimal inequalities, i.e., inequalities that are fulfilled by at least one of the dual optimal solutions and can thus be used to reduce the search space (see Valério de Carvalho [40] and Ben Amor et al. [8] for further details on this type of techniques). We also mention that Caprara et al. [13] solved the TKP by using a so-called recursive Dantzig-Wolfe reformulation, which uses the reformulation not only for solving the original master problem, but also for recursively solving the pricing sub-problems.

3. Mathematical models

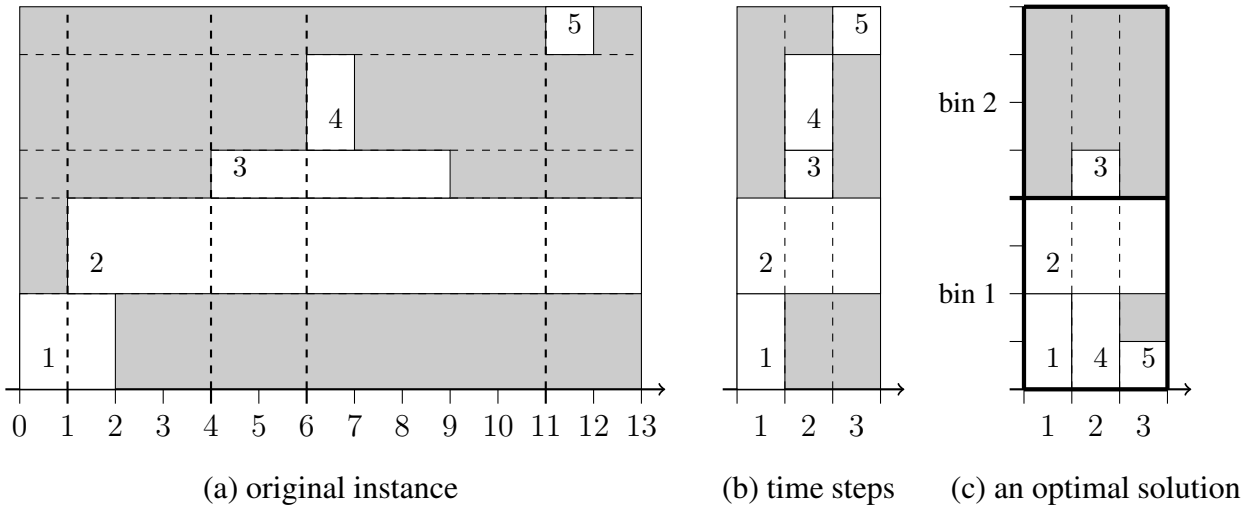
In order to derive suitable models for TBPP, we start by showing that the TBPP can be modeled by considering only a polynomial number of time instants. Despite the capacity requirements being defined on the entire time horizon, a weight variation may arise only at the starting time of an item. Therefore, it is sufficient to satisfy the capacity restrictions at the n starting times of the items. Given an item $j \in N$, let us define $\overline{S}_j := \{\ell \in N : s_\ell \leq s_j \text{ and } t_\ell > s_j\}$ as the set of active items at time s_j (note that $j \in \overline{S}_j$). As all items in $\overline{S}_j$ are active at the same time instant, a capacity constraint must be imposed for these items. Moreover, if $\overline{S}_j \subseteq \overline{S}_k$, then the associated capacity constraint at time s_j is dominated by that of time s_k. Let us define $\overline{T} = \{t \in N : \overline{S}_t \nsubseteq \overline{S}_k, \forall k \in N\}$ as the index set of all the *non-dominated* constraints (or non-dominated sets). To model the problem, it is enough to consider the capacity usage at each $t \in \overline{T}$.

To simplify the presentation we compact the indices in $\overline{T}$ by shifting them into the set $T = \{1, \ldots, |\overline{T}|\}$ and we rename the corresponding non-dominated sets as $S_1, \ldots, S_{|T|}$. We call $t \in T$ a *time step*. W.l.o.g., let us also suppose that the items are sorted by non-decreasing starting times.

In Figure 1, we give an example to illustrate a TBPP instance with five items having weights

$w_1 = w_2 = w_4 = 2, w_3 = w_5 = 1$, and bin capacity $W = 4$. The starting time and ending time (s_j and t_j) are shown in Figure 1(a). For instance, item 1 is active in the first two time instants and item 2 is active from the second to the last time instant. The simultaneously active sets of items are : $\overline{S}_1 = \{1\}$, $\overline{S}_2 = \{1, 2\}$, $\overline{S}_3 = \{2, 3\}$, $\overline{S}_4 = \{2, 3, 4\}$ and $\overline{S}_5 = \{2, 5\}$. The non dominated sets are $\overline{S}_2$, $\overline{S}_4$ and $\overline{S}_5$. In Figure 1(b) we show the renumbered non-dominated sets $S_1 = \{1, 2\}$, $S_2 = \{2, 3, 4\}$ and $S_3 = \{2, 5\}$. Finally, Figure 1(c) reports an optimal TBPP solution using two bins, the first containing items 1, 2, 4, 5 and the second only item 3.

One can also use the example to note the difference between TBPP and 2D-BPP. Consider an instance of 2D-BPP defined by the rectangles in Figure 1, and 3×4 rectangular bins. An optimal 2D-BPP solution packs item 3, together with all the other items, in the first bin. This derives from the fact that in the 2D-BPP the rectangles can be shifted not only vertically (as in the TBPP) but also horizontally (because there are no time windows), so a single bin is enough to solve the instance.



(a) original instance (b) time steps (c) an optimal solution

Figure 1: Example of a TBPP instance with 5 items and 3 time steps.

We can now show that the TBPP is a special case of the VPP. Given a TBPP instance, define a corresponding VPP instance with the same items and $|T|$ dimensions each with capacity W. For each item $j \in N$ define the item weights as

$$w_{jt} = \begin{cases} 0 & \text{if } t < s_j \text{ or } t \geq t_j \\ w_j & \text{otherwise} \end{cases} \qquad t \in T. \tag{1}$$

One can see that any VPP solution to this instance is a feasible solution for the TBPP instance, and vice versa, so the VPP generalizes the TBPP.

3.1. A polynomial-size model

In this section, we introduce the first *Integer Linear Programming* (ILP) formulation for the TBPP. Let $I = \{1, 2, \ldots, m\}$ be the set of bins, where $m \leq n$ is an upper bound on the number of bins necessary to pack all items. We introduce a set of binary variables y with the following meaning:

$$y_i = \begin{cases} 1 & \text{if bin } i \text{ is used,} \\ 0 & \text{otherwise,} \end{cases} \qquad i \in I;$$

and a second set of binary variables x such that:

$$x_{ij} = \begin{cases} 1 & \text{if item } j \text{ is packed in bin } i, \\ 0 & \text{otherwise,} \end{cases} \qquad i \in I, j \in N.$$

The polynomial-size ILP formulation, called ILP^c in the reminder of the paper, reads as follows:

$$\min \sum_{i \in I} y_i \tag{2}$$

$$\sum_{i \in I} x_{ij} = 1 \qquad j \in N, \tag{3}$$

$$\sum_{j \in S_t} w_j x_{ij} \leq W y_i \qquad i \in I, t \in T, \tag{4}$$

$$x_{ij} \in \{0,1\} \qquad i \in I, j \in N, \tag{5}$$

$$y_i \in \{0,1\} \qquad i \in I. \tag{6}$$

The objective function (2) minimizes the number of used bins, constraints (3) impose that each item is packed in one bin, and constraints (4) impose that for each bin and for each time step the total weight of the active items does not exceed the bin capacity. The optimal solution value of ILP^c is denoted by $z(\text{ILP}^c)$, we use the same notation for all the other models.

3.2. An exponential-size model

In this section, we describe a second formulation for the TBPP characterized by an exponential number of variables associated with all feasible packing patterns, i.e., subsets of items respecting the bin capacity at any time step. Let $\mathscr{P}$ represent the collection of all feasible packing patterns:

$$\mathscr{P} = \left\{ P \subseteq N : \sum_{j \in S_t \cap P} w_j \leq W, \quad t \in T \right\}.$$

For each pattern $P \in \mathscr{P}$, we introduce a binary variable ξ_P with the following meaning:

$$\xi_P = \begin{cases} 1 & \text{if packing pattern } P \text{ is selected,} \\ 0 & \text{otherwise.} \end{cases} \qquad P \in \mathscr{P}.$$

The exponential-size ILP formulation, called ILP^e in the reminder of the paper, reads as follows:

$$\min \sum_{P \in \mathscr{P}} \xi_P \tag{7}$$

$$\sum_{P \in \mathscr{P}: j \in P} \xi_P = 1 \qquad j \in N, \tag{8}$$

$$\xi_P \in \{0,1\} \qquad P \in \mathscr{P}. \tag{9}$$

The objective function (7) minimizes the number of packing patterns (bins) used, and constraints (8) ensure that each item is packed in one bin.

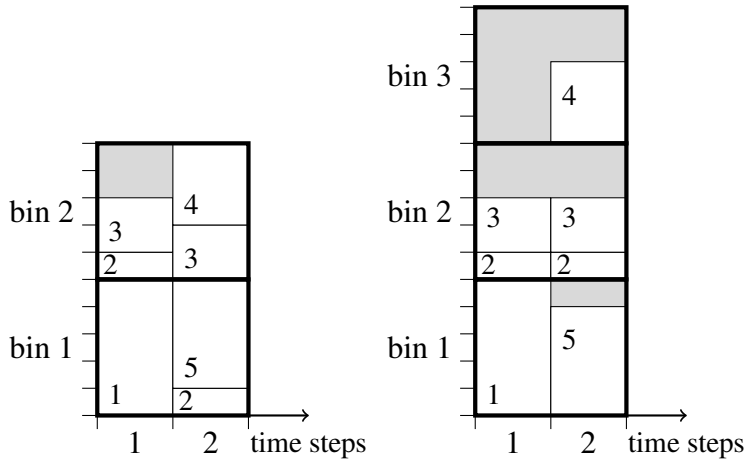
4. Lower bounds

We introduce some lower bounding techniques which are useful to solve several instances to proven optimality in short computing times.

Observe that for a single time step $t \in T$, the ILP^c models the BPP by considering only the items in S_t. We denote this formulation as ILP$^c(t)$. A valid lower bound for the TBPP is thus:

$$LB_0 = \max_{t \in T} z(\text{ILP}^c(t)). \tag{10}$$

The following example shows that the optimal solution value of the TBPP may be strictly greater than the optimal solution value of the ILP$^c(t)$ for each $t \in T$. Consider five items with weights $w_1 = 10, w_2 = 2, w_3 = 4, w_4 = 6, w_5 = 8$, a bin capacity $W = 10$ and two time steps defined by sets $S_1 = \{1, 2, 3\}$ and $S_2 = \{2, 3, 4, 5\}$. In Figure 2(a) we show that the optimal ILP$^c(t)$ solutions, for $t = 1, 2$, use two bins, but the optimal TBPP solution uses three bins, as depicted in Figure 2(b). Two bins can be obtained if and only if items 2 and item 3 are packed in different bins in the two time steps. As this does not lead to a feasible TBPP solution, three bins are necessary for an optimal TBPP solution.



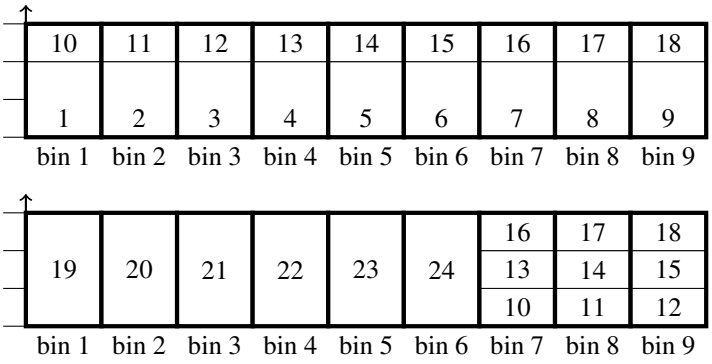
(a) optimal ILP$^c(t)$ solutions (b) a TBPP optimal solution

Figure 2: A small example with $LB_0 < z(\text{ILP}^c)$

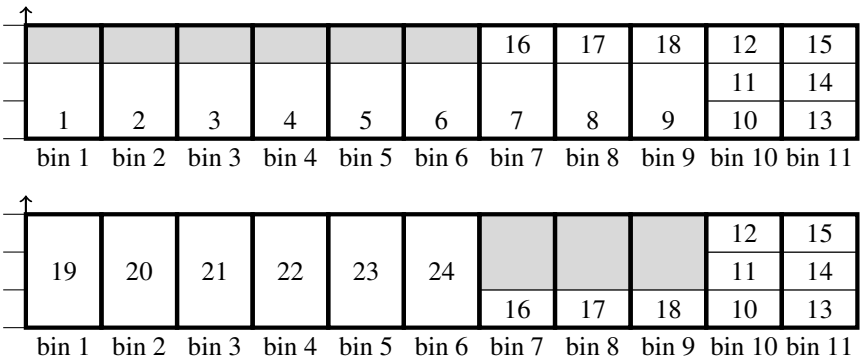
The difference between LB_0 and the optimal TBPP solution value can be greater than one, as shown in the following example. Let $n = 24$, $W = 3$ and item weights $w_j = 2$ ($j = 1, \ldots, 9$), $w_j = 1$ ($j = 10, \ldots, 18$) and $w_j = 3$ ($j = 19, \ldots, 24$). Two are the time steps and the corresponding sets of simultaneously active items are defined as follows: $S_1 = \{1 \leq j \leq 18\}$ and $S_2 = \{10 \leq j \leq 24\}$.

The optimal ILP$^c(t)$ solution for $t = 1, 2$ has 9 bins as shown in Figure 3(a) (note that, to reduce space, this figure is drawn with a 90^o rotation with respect to the previous ones, i.e., the bins are on the horizontal axis and the time steps in the vertical axis). The optimal TBPP solution has 11 bins (Figure 3(b)), hence two units larger than LB_0.

Let us now consider the continuous relaxation of ILP^c and ILP$^c(t)$, denoted, respectively, by LP^c and LP$^c(t)$. A second valid TBPP lower bound is given by $z(\text{LP}^c)$. For any $t \in T$, it is known



(a) Optimal ILP$^c(t)$ solutions for time steps 1 (above) and 2 (below)



(b) Optimal TBPP solution: time steps 1 (above) and 2 (below)

Figure 3: Example showing that LB_0 may have a gap larger than one.

(see e.g. Martello and Toth [35]) that the optimal solution value of LP$^c(t)$ can be computed as: $z(\text{LP}^c(t)) = \frac{1}{W}\sum_{j\in S_t} w_j$, from which we immediately have

$$z(\text{LP}^c) = \max\{z(\text{LP}^c(t)), t \in T\} = \frac{1}{W}\max\left\{\sum_{j\in S_t} w_j, t \in T\right\}.$$

Given an instance of the TBPP let $\tilde{t} = \text{argmax}\{\sum_{j\in S_t} w_j : t \in T\}$, then a lower bound from the continuous relaxation of ILP^c is

$$LB_1 = \lceil z(\text{LP}^c(\tilde{t}))\rceil. \tag{11}$$

A straightforward implementation of LB_1 requires an $O(n^2)$ computing time, but this bound can be improved as shown in the next property.

Property 1. *Given an instance of the* TBPP*, lower bound LB_1 can be computed in $O(n \log n)$ time.*

Proof. Remind that the items are sorted by non-decreasing starting times. Compute a list $\mathcal{L}$ containing the items sorted by non-decreasing ending time t_j. Start with a total item weight $\mathcal{W} = 0$ and consider the items one at a time, in the original order. For each item j, add w_j to $\mathcal{W}$ and remove from list $\mathcal{L}$ each item k with $t_k \leq s_j$, by reducing $\mathcal{W}$ of the corresponding weights w_k. Store the

maximum value of $\mathcal{W}$ obtained and let $\tilde{t}$ denote the corresponding time step. The initial sorting can be implemented in $O(n \log n)$, while the scanning of the items requires $O(n)$ because each item is considered exactly once in each list, and the thesis follows. □

Another lower bound, which requires an intermediate computing effort between that of LB_0 and LB_1, can be obtained by solving a single BPP on time step $\tilde{t}$, thus obtaining:

$$LB_0(\tilde{t}) = z(\text{ILP}^c(\tilde{t})).$$

4.1. Preprocessing with item weight lifting

In order to improve lower bound LB_1, see (11), we can apply lifting techniques (see, e.g., Dell'Amico et al. [18]) which try to increase the item weights, while ensuring that the lifted instance has the same optimal solution value of the original one.

Given an item $j \in N$, let $\gamma(j) = \{k \in N \setminus \{j\} : \exists t \in T \text{ such that } j, k \in S_t\}$ denote the set of items that are active in at least one time step where j is active. We can use the following ILP model to compute the maximum possible loading of a bin where item j is packed:

$$\sigma(j) = \max \left\{ \sum_{k \in \gamma(j)} w_k x_k : \sum_{k \in \gamma(j)} w_k x_k \leq W - w_j, x_k \in \{0,1\}, k \in \gamma(j) \right\}. \tag{12}$$

Given the optimal value $\sigma(j)$ (or any valid upper bound on $\sigma(j)$) we know that the bin where j is packed, in the time steps where j is active, has at least an empty space $W - w_j - \sigma(j)$, so we can lift the weight of j as:

$$w_j = w_j + (W - w_j - \sigma(j)) = W - \sigma(j). \tag{Lift-1}$$

Model (12) represents a Subset Sum Problem (SSP) (see, e.g., Martello and Toth [35]). To implement this lifting procedure we have to execute a first step which solves n SSPs to try to lift the item weights. Once an item j has been lifted, a new iteration can be executed to try to further lift the items in $\gamma(j)$.

Another lifting procedure can be obtained by considering each time step $t \in T$, such that $j \in S_t$, and defining the following SSP:

$$\sigma(j,t) = \max \left\{ \sum_{k \in S_t \setminus \{j\}} w_k x_k : \sum_{k \in S_t \setminus \{j\}} w_k x_k \leq W - w_j, x_k \in \{0,1\}, k \in S_t \setminus \{j\} \right\}. \tag{13}$$

Similarly to the previous case, we see that, for the given time step t, we can lift the weight of the item j to $W - \sigma(j,t)$. By considering all the time steps in which j is active, we define the valid lifting:

$$w_j = n_{t \in T : j \in S_t}(W - \sigma(j,t)). \tag{Lift-2}$$

The implementation of this lifting requires to solve $n \times |T|$ SSPs for each iteration.

Property 2. *For a given item $j \in N$, lifting Lift-2 dominates lifting Lift-1.*

Proof. Let $\tilde{t}$ denote the time instant giving the minimum value in (Lift-2). The thesis immediately follows because $S_{\tilde{t}} \subseteq \gamma(j)$. □

We note that, although observation 2 guarantees that Lift-2 dominates Lift-1 for a given item, it does not guarantee that the total lift on all the items obtained by Lift-2 dominates that obtained by Lift-1.

In the following, we will denote as LB_2^I and LB_2^{II} the value of LB_1 computed with weights lifted using the first and second method, respectively, and a single lifting iteration.

4.2. Lower bound from the exponential-size formulation

Another lower bound can be obtained by computing the continuous relaxation of the exponential-size formulation ILP^e, that we denote as LP^e in what follows. We have thus

$$LB_3 = \lceil z(\text{LP}^e) \rceil. \tag{14}$$

The following property states that the quality of the lower bound obtained solving the LP relaxation of ILP^c is dominated by its counterpart associated with ILP^e.

Property 3. *The optimal value of* LP^e *is greater than or equal to the optimal value of* LP^c.

Proof. We start the proof by showing that any feasible solution of LP^e can be transformed into a feasible solution of LP^c preserving its objective function value. W.l.o.g., we consider the case in which $m = n$. Any optimal basic solution of LP^e can have at most n non-zero variables; let ξ^* denote a feasible solution of LP^e, and $P(i)$ be a function that returns the i-th active pattern in ξ^* (i.e., $P(i)$ is the i-th pattern associated to a strictly positive variable). We can define a solution (x^*, y^*) of LP^c as follows: for each bin $i \in I$ and for each $j \in N$, we can set:

$$y_i^* = \xi_{P(i)}^* \quad \text{and} \quad x_{ij}^* = \xi_{P(i)}^*.$$

In case $\sum_{i \in I} x_{ij}^* > 1$, for some j, we can arbitrarily reduce the x_{ij}^* in order to sum up 1. By construction, the solution (x^*, y^*) is feasible for LP^c and has the same objective function value.

We then show a case where the optimal value of LP^e is strictly larger than the optimal value of LP^c. Consider the instance presented in Figure 2 with 5 items and 2 time steps. The optimal solution value of LP^c is equal to 2. This optimal value can be obtained for instance by the solution $y_1^* = y_2^* = 1$ and $x_{ij} = \frac{1}{2}$ for $i = 1, \ldots, 5$ and $j = 1, 2$ (all the other LP^c variables are set to zero). An optimal solution for LP^e is defined by the four feasible packing patters $P_1 = \{2, 3\}$, $P_2 = \{2, 4\}$, $P_3 = \{1, 5\}$, $P_4 = \{3, 4\}$, and by the corresponding variables $\xi_{P_1} = \xi_{P_2} = \xi_{P_4} = 0.5$ and $\xi_{P_3} = 1$. The optimal solution value is $z(\text{LP}^e) = 2.5 > z(\text{LP}^c) = 2$. □

5. Upper bounds

We start by describing some simple heuristic algorithms that can be used to provide approximate solutions in short computing times.

Greedy algorithm. The first method we introduce is based on a sequence of greedy algorithms derived from the well known *First-Fit* algorithm for BPP (see, e.g., Johnson [31]). The First-Fit performs n iterations by considering one item at a time, in a given order. At the beginning no bin is open (used). In the first iteration, a single bin is opened and the first item is packed in it. At each iteration $j > 1$, the algorithm looks for the first open bin in which j fits, if any. If this bin exists j is packed in it, otherwise a new empty bin is opened and used to pack j. The algorithm runs in $O(n^2)$ time.

We extend the First-Fit for the BPP to the TBPP, by simply checking, for each item j and tentative bin i, if packing j in i is feasible for all the time steps in $[s_j, t_j)$. The time complexity increases to $O(n^2|T|)$, but in practice these methods are very fast. We implemented two versions: in First-Fit-1 we do not make any sorting of the items, but we consider them in their natural order given by increasing s_j values. In First-Fit-2 we sort the items by non-decreasing w_j values. The two versions were run on the original instances and on those lifted with Lift-1 (see Section 4.1). In the following, we call UB_1 the best solution value obtained by running First-Fit-1 and First-Fit-2 on the original and lifted instances.

Rolling horizon heuristic. We developed a more powerful heuristic using a rolling horizon concept. Let $\Delta \in [1, \ldots, |T|]$ denote the width of a heuristic time window. Our *rolling horizon heuristic* performs $\lceil\ |T|/\Delta\ \rceil$ iterations, and at each iteration it solves model ILP^c with a limited number of time-step constraints. More specifically, let ILP$^c{}_{k\Delta}$, with $k = 1, \ldots, \lceil\ |T|/\Delta\ \rceil$, denote the restricted ILP^c model obtained by: (i) defining constraints (4) only for time steps in $T_k = [(k-1)\Delta + 1, n(k\Delta, |T|)]$, and (ii) selecting only the x variables associated with the items in $N_k = \cup_{t \in T_k} S_j$ (i.e., the active items in T_k). If a feasible solution for ILP$^c{}_{k\Delta}$ is found, we pack the items in N_k according to this solution, by fixing, the corresponding x variables to one. These items are no longer considered for possible reassignment. Due to the variable fixing the next model ILP$^c{}_{(k+1)\Delta}$ starts from the packing of the items in $\cup_{h=1}^{k} N_h$ and packs the new items from N_{k+1} in the residual space of the opened bins, or in new bins. If for some iteration k no feasible solution to ILP$^c{}_{k\Delta}$ is found, the algorithm terminates with no TBPP solution. We will call UB_2 the solution value obtained by the rolling horizon heuristic.

A further upper bound UB_3 has been developed using a truncated version of the exact branch-and-price algorithm of Section 6. We will give details on this heuristic in that section.

6. Solving the Exponential-Size formulation

In Section 3, we introduced a polynomial-size formulation ILP^c and an exponential-size formulation ILP^e. Formulation ILP^c can be explicitly written, also for large size instances, and solved using a generic ILP solver. Formulation ILP^e, instead, has exponentially many variables that cannot be explicitly enumerated for large-size instances. *Column Generation* (CG) techniques are then necessary to efficiently solve the continuous relaxation of ILP^e (we refer the interested reader to, e.g., Desaulniers et al. [24] for further details on column generation). In the following, we present a new branch-and-price framework for ILP^e. Two are the main ingredients of a branch-and-price algorithm: (i) a column generation algorithm to solve the Linear Programming Relaxation of the

exponential-size integer model, and (ii) a branching scheme. We discuss separately these two aspects in the next sections, before introducing a heuristic based on this framework, and our final overall algorithm for the TBPP.

6.1. Solving the Linear Programming Relaxation of ILP^e

Model LP^e, initialized with a subset of variables (columns) defining feasible solutions, is called the *Restricted Master Problem* (RMP). In our implementation, we initialize the RMP with the columns associated with the best solution provided by the Greedy algorithm of Section 5. Its solution provides a (sub)-optimal primal solution. To find the optimal solution we consider the dual of this LP:

$$\max \left\{ \sum_{j \in N} \pi_j : \quad \sum_{j \in P} \pi_j \leq 1,\ P \in \mathscr{P}, \quad \pi_j \geq 0,\ j \in N \right\}, \tag{15}$$

where π_j is the dual variable associated with the j-th constraint (8). It is worth noticing that we use the '$\geq$' sign in constraint (8), instead of the '=' sign, to have non negative dual variables. This choice does not change the problem, since any solution of (7)-(9) which packs an item in more than one bin can be transformed into a TBPP solution by arbitrarily removing the item from all the used patterns, but one. A violated dual constraint induces a negative reduced cost in the primal problem, so the corresponding primal variable must be added to the RMP to find an optimal solution. Accordingly, the column generation performs a number of iterations where violated dual constraints are added to the RMP in form of primal variables, and the RMP is re-optimized, until no violated dual constraint exists. At each iteration, the so-called *Pricing Problem* is solved. This problem asks to determine (if any) a packing pattern $P^* \in \mathscr{P}$ for which the associated dual constraint is violated, i.e., such that

$$\sum_{j \in P^*} \pi_j^* > 1, \tag{16}$$

where π^* is the optimal vector of dual variables for the current RMP.

If a packing pattern P^* has dual weight larger than one (that is, the reduced cost is negative), the associated column is added to the RMP and the problem is re-optimized. If, on the other hand, the dual weight is not larger than 1, by linear programming optimality conditions no column can improve the objective function of the RMP and therefore LP^e is solved to optimality.

In the solution of the pricing problem we want to find a violating pattern P^*, or to prove that no one exists. The following 0-1 *Temporal Knapsack Problem* (TKP) models the separation using a binary variable z_j that takes value 1 if and only if item $j \in N$ is selected in subset P^*:

$$z_s(\pi^*) = \max \left\{ \sum_{j \in N} \pi_j^* z_j : \quad \sum_{j \in S_t} w_j z_j \leq W,\ t \in T, \quad z_j \in \{0,1\},\ j \in N \right\}. \tag{17}$$

If $z_s(\pi^*) > 1$ a violating pattern $P^* = \{j \in N : z_j = 1\}$ has been found, otherwise the RMP solution is optimal.

6.2. *Branching schemes for* ILP^e

The design of a branching scheme is crucial for the performance of a branch-and-price algorithm (see, e.g., Vanderbeck [41]). In the following, we describe two branching scheme adopted in our new branch-and-price framework. Two are the main properties that a branching rule should hold. Firstly, it is a complete scheme, i.e., it ensures that integrality can be imposed in all cases. Secondly, it does not require modifications to the master problem and it does not impact much on the pricing algorithm. The latter property means that an ideal branching does not alter the structure of the pricing problem so that the same algorithm can be applied during the entire search. In the following, we denote with ξ^* a fractional solution to LP^{e}at a given node of the branching tree, and with $\widehat{\mathscr{P}} \subseteq \mathscr{P}$ the set of columns in the RMP at the node.

Branching BR-1. This is the standard branching rule, which selects a variable ξ^*_P with fractional value and separates the current node into two children nodes, by imposing, respectively, $\xi^*_P = 1$ and $\xi^*_P = 0$. The advantage of this rule is to force the algorithm to find feasible solutions in short time, in particular when a deep-first exploration rule is used and the left branch (with $\xi^*_P = 1$) is selected first. The RMP can easily incorporate the branching constrains, because one can directly impose the branching variable value. The pricing on the left branch is easy, because we can just remove the items in P from the pricing problem, but on the right branch it is necessary to add a cut to avoid generating P again. To achieve this, we use the cut

$$\sum_{j \in N \setminus P} z_j - \sum_{j \in P} z_j \geq 1 - |P|. \tag{18}$$

Branching BR-2. The second branching strategy we propose is inspired by the *Rayan-Foster branching* scheme and it preserves the pricing algorithm in part of the branching nodes. This rule is designed to impose that each couple of items are either packed in the same bin or in different ones. This rule has been proposed for branch-and-price algorithms based on set-covering formulations (see, e.g., Barnhart et al. [5]) and used to derive several effective exact algorithms for the BPP, see e.g., Wei et al. [43]. A couple of items r and $s \in N$ is *fractionally packed* if:

$$\sum_{P \in \widehat{\mathscr{P}}: r,s \in P} \xi^*_P = \gamma, \tag{19}$$

with γ fractional. In case more than one pair of fractionally packed items exist, we select the couple r and s associated with the first fractional γ value. Two children nodes are then created:

- in the first node we force items r and s to be packed in the same bin;
- in the second node we force items r and s to be packed in different bins.

This branching rule can be implemented without any additional constraint to the RMP, indeed it is enough to remove from RMP the columns that does not respect the rule. For the pricing problem we can see that in the left branch we can force a couple (r, s) of items to be packed together by replacing the couple with a *super item*, say $\hat{\jmath}$ with $s_{\hat{\jmath}} = n(s_r, s_s)$, $t_{\hat{\jmath}} = \max(t_r, t_s)$ and weight

depending on the time steps:

$$w_{\hat{\jmath}t} = \begin{cases} w_r & \text{if } r \in S_t,\ s \notin S_t, \\ w_s & \text{if } r \notin S_t,\ s \in S_t, \\ w_r + w_s & \text{if } r \in S_t,\ s \in S_t, \end{cases} \qquad t \in [s_{\hat{\jmath}}, t_{\hat{\jmath}}].$$

The pricing problem is modified by substituting the capacity constraints in (17) with constraints $\sum_{j \in S_t} w_{jt} z_j \leq W, t \in T$, so the problem remains a TKP but with time step dependent item weights.

In the right branch we enforce the two items to belong to different patterns. In this case the pricing problem (17) must be changed by introducing the so called *conflicts* between items. Unfortunately the conflicts cannot be directly managed with some modification of the input instance, and we are forced to solve the problem as a generic MIP with the additional constraint $z_r + z_s \leq 1$.

The branching rule BR-1 is clearly complete because we have no variable to branch only when all values are integer, and the solution is integer. The following observation states that also the branching rule BR-2 is complete for ILP^e:

Property 4. *The branching rule BR-2 provides a complete branching scheme for model* ILP^e.

Proof. If at each node of the branching tree we select a pair of items and fix them to be packed in the same bin or in separate bins, then, after at most $O(n^2)$ branchings, all couples have been fixed and all solutions enumerated. To prove the thesis, it is enough to show that we can always find two items providing a fractional γ value, as defined in (19).

In Barnhart et al. [5], it is proved that for any 0-1 constraint matrix A (as for the case of LP^e), if a basic solution ξ^* to $A\xi = 1$ is fractional, then there exist two rows r and s such that:

$$0 < \sum_{P \in \mathscr{P} \,:\, r,s \in P} \xi^*_P < 1. \tag{20}$$

As we adopted a covering formulation ($A\xi \geq 1$), we can use the above property to prove that the required items exist if ξ^* determines at least two strict constraints (i.e., it is satisfied with the "=" sign). We show that these two rows always exists by analyzing all possible cases. We first note that in an optimal solution there must be at least one strict constraint, otherwise it is possible to improve the solution by reducing the value of one variable, until the l.h.s. of one row takes the value one. If there is at least another strict constraint we are done, otherwise let r denote the row of the unique strict constraint. If there is a packing P not containing item r and such that $\xi^*_P > 0$, we can improve the solution by reducing ξ^*_P until a second constraint is strict, and this case is closed. Otherwise let $\widehat{\mathscr{P}}^+$ denote the subset of columns of $\widehat{\mathscr{P}}$ with a positive ξ^* value. In the remaining case, $r \in P$ for all $P \in \widehat{\mathscr{P}}^+$, and for any $s \in N \setminus \{r\}$ equation (19) defines an integer γ value (otherwise we have found the r, s couple to be used for branching). But the latter case is not possible because it implies that any item is packed within r in all columns of $\widehat{\mathscr{P}}^+$, which results to be identical. □

6.3. Exponential-size formulation based heuristic

Effective heuristic algorithms can be obtained by contaminating a branch-and-price framework with one or more heuristic rules that cut parts of the search tree. The *diving* metaphor in an LP-based

branch-and-bound tree foresees a search that plunges deep into the enumeration tree by selecting a branch with some heuristic rule at each node.

In our case, we have implemented the branch-and-price with branching rule BR-1 (which fixes a variable ξ_P at a time to 1 and 0, respectively), but we have limited the number of possible branching using a *token-based* rule. Fixing a variable at value 1 (i.e., fixing the packing of a bin) reduces the problem size and drives the algorithm to find a feasible solution in a short time. For this reason, we allow the algorithm to perform all left branches (which fix the variable to 1), but we restrict the number of right branches. More specifically, at the beginning the algorithm has K token available for right branching. When a tree node k is separated using a right branch, the number of available tokens is reduced by one. When no more token exist, the algorithm can perform only left branches. If backtracking occurs and the search return to node k the token possibly used for a right branch returns available and the total number of tokens is increased by one. One can see that with $K \geq 1$ tokens at most $O(n2^{(K-1)})$ solutions are generated, while a single solution is generated for $K = 0$.

6.4. Overall algorithm

The final algorithm we propose to solve the TBPP is based on a combination of the above approaches. In particular, in a first phase, we start by computing the lower bound LB_0 and an upper bound, say UB_0, obtained by running the Greedy and the H-Rolling heuristics, and choosing the best solution. If $UB_0 = LB_0$, then we stop. Otherwise, in the second phase, we compute the continuous relaxation LP^e. We check again if UB_0 provides the optimal solution by comparing it with LB_3, and stopping if equality holds. In the third phase, we run the tree-exploration of the H-Diving heuristic, starting from the continuous relaxation LP^e already computed while evaluating LB_3. Let UB_3 be the corresponding heuristic solution value. If $UB_3 = LB_3$ we are done, otherwise we proceed to the final phase running the branch-and-price method of Section 6, using branching rule BR-2. The algorithm, called B&P$^+$ in the following, is resumed in Algorithm 1. We observe that phase 2 is one of the most time consuming, because it requires to solve LP^e using the column generation approach. Once LP^e is solved the next tree-search performed by H-Diving with a single token, is usually fast. In phase 4, instead, the tree-search using branching rule BR-2 without restrictions, may use large computing times. See Section 7 for details.

Algorithm 1: B&P$^+$

1: Compute $L = \max(LB_0, LB_1, LB_2)$; ▷ Phase 1
2: Run First-Fit-1 and First-Fit-2 on the original and lifted instance, run H-Rolling, and let U be the minimum solution value;
3: **if** $U = L$ the solution is optimal **then return**;
4: Compute $L = \max(L, LB_3)$; **if** $U = L$ the solution is optimal **then return**; ▷ Phase 2
5: Run H-Diving with one token and possibly improve U; ▷ Phase 3
6: **if** $U = L$ the solution is optimal **then return**;
7: Run the branch-and-price with branching rule BR-2, possibly improving L and U ▷ Phase 3
8: **return**.

7. Computational Results

The experiments have been performed on a computer with a 3.10 GHz 4-core Intel Xeon processor and 16Gb RAM, running a 64 bits Ubuntu Linux operating system version 14.04.5. The algorithms were coded in `C++` and all the codes were compiled with `gcc 6.2` and `-O3` optimizations. To solve the linear relaxations of the models and the required ILPs, we used `Cplex 12.7`, run on a single-thread (parameter `CPX_PARAM_THREADS` set to one).

We build our testbed starting from the TKP instances called "I" in Caprara et al. [11]. The testbed "I" consists of one hundred instances which are further divided into ten classes generated using different values of some inout parameters (see Table 1), and the following rules, which allows to generate only the inclusion-wise non-dominated sets of simultaneously active items (see Caprara et al. [11] for further details). Given the size $|T^o|$, for each $t \in \{1, \ldots, |T^o|\}$ the number of tasks in S_t is uniformly distributed in $[a_n, a_{\max}]$. If $t > 1$ then β percent tasks from S_{t-1} are randomly selected and inserted in S_t, where β is uniformly distributed in $[b_n, b_{\max}]$. Item weights are uniform random integer from $[10, 100]$. The bin capacity is $W = 100$ for all instances. For Classes I-IV, $|T^o| = 2688 + 128(i-1)$ $(i = 1, \ldots, 10)$. For Classes V-X, $|T^o| = 768 + 128(i-1)$ $(i = 1, \ldots, 10)$.

Table 1: Parameter values used to generate the test instances

Class	a_n	$a_{\max}$	b_n	$b_{\max}$	Class	a_n	$a_{\max}$	b_n	$b_{\max}$
I	10	10	90	95	VI	30	30	70	90
II	15	15	90	95	VII	30	30	90	95
III	20	20	90	95	VIII	25	35	90	95
IV	25	25	90	95	IX	25	35	70	90
V	30	30	90	95	X	30	40	90	95

Due to the fact that the TBPP instances share the same data structure with those of the TKP, except for the absence of item profits, these instances could be directly used to define TBPP instances. However the BPP is normally much harder to solve than a KP, and the same happens with the time versions here considered. Therefore we reduced the size of the original instances as follows. Let T^o denote the time steps in an original instance, we build TBPP instances with $|T| \in \{5, 10, 15, 20, 30, \ldots, 150\}$ by extracting from the original instance all the items in $\cup_{t=1}^{|T|} S_t$, and disregarding the profits.

Instances of classes VI and IX have b_n smaller than the other classes. This implies that when generating a set S_t a smaller number of tasks are taken from S_{t-1}, and thus a larger number of new tasks exist in S_t. This fact, however, does not translate into a greater difficulty of the class, as will be seen in the remaining of the section.

7.1. Lower bounds

Our first lower bound LB_0 (see Section 4) requires to solve $|T|$ BPP instances. To solve the BPPs we implemented the arc-flow algorithm proposed in Valério de Carvalho [39]. Lower bound LB_1 can be computed in $O(n \log n)$ as shown by Property 1, while LB_2^I requires to solve n Subset Sum problems for each iteration (see Section 4.1). We executed a single iteration by solving the

SSP with Cplex. The next lower bound LB_2^{II} is still based on a lifting procedure, and requires the solution of $O(n|T|)$ SSPs for each iteration. Again we used a single iteration and Cplex for solving the subproblems.

Table 2 reports the results of the lower bounds for instances with $|T| = (10, 20, \ldots, 100)$. Each row corresponds to 100 instances from the ten classes (ten instances for each class). The column labeled $|T|$ gives the number of time steps in the 100 instances, the column labeled '$|N|$' gives the average number of items. Columns '# opt', 'avg gap' and 'max gap', report, respectively, the number of times the bound is equal to the optimal solution value and the average and maximum absolute gap with respect to the optimum. (The optimal solution values are computed with the exact methods evaluated in the next Section 7.3, possibly running them for long times, when no method is able to produce a proven optimum within the given time limit.). Column 'time' reports the average computing time, when it is not negligible. The last row of the table reports the total number of optimal values found and the average gap and time, for each lower bound. We do not report on LB_2^{II} since it does not improves upon LB_2^I, and uses some more computing time. Table 3 gives the same information grouped by instance class: in this case each row refers to 150 instances.

		LB_0				$LB_0(\tilde{t})$			LB_1			LB_2^I			LB_3	
			avg	max	avg		avg	max		avg	max		avg	max		avg
$\|T\|$	$\|N\|$	# opt	gap	gap	time	# opt	gap	gap	# opt	gap	gap	# opt	gap	gap	# opt	time
10	54.90	97	0.03	1	0.08	92	0.08	1	12	1.49	4	22	1.20	4	100	0.24
20	88.39	96	0.04	1	0.15	88	0.13	2	3	1.82	4	12	1.46	4	100	0.90
30	121.43	95	0.05	1	0.21	83	0.18	2	2	2.02	4	8	1.66	4	100	2.23
40	154.10	96	0.04	1	0.29	83	0.18	2	2	2.09	4	6	1.72	4	100	4.23
50	186.70	98	0.02	1	0.36	83	0.21	3	1	2.31	5	3	1.90	4	100	5.51
60	219.61	97	0.03	1	0.45	82	0.22	3	1	2.27	5	3	1.89	4	100	9.80
70	252.50	98	0.02	1	0.52	78	0.26	3	1	2.31	5	3	1.93	4	100	14.41
80	285.93	98	0.02	1	0.57	82	0.20	3	1	2.40	5	5	1.95	4	100	18.07
90	318.86	98	0.02	1	0.62	79	0.24	3	1	2.49	5	2	2.04	4	100	30.14
100	351.78	99	0.01	1	0.69	77	0.27	3	1	2.53	5	2	2.10	4	100	44.61
110	385.04	98	0.02	1	0.76	79	0.27	3	1	2.55	5	1	2.10	4	100	29.79
120	417.73	99	0.01	1	0.84	80	0.27	3	0	2.61	5	0	2.15	4	100	38.23
130	451.12	99	0.01	1	0.92	78	0.30	3	0	2.58	5	0	2.14	4	100	48.40
140	483.61	99	0.01	1	0.97	79	0.29	3	0	2.59	5	1	2.13	4	100	65.85
150	516.45	99	0.01	1	1.03	79	0.30	3	0	2.61	5	2	2.14	4	100	67.47
		1466	**0.02**		**0.56**	**1222**	**0.23**		**26**	**2.31**		**70**	**1.90**		**1500**	**25.33**

Table 2: Performance of the Lower Bounds

Bound LB_0 is quite effective and fails to find the optimum solution value in only 34 instances over 1500. Its performances improve with the problem size. Bound $LB_0(\tilde{t})$ fails on 278 instances and its performances definitely worsens when the size of the instance increases. Bounds LB_1 and LB_2^I are not competitive since they are able to provide very few optimal values. The best bound is LB_3 which is able to give the optimal value for all the 1500 instances. However, this result is obtained with a larger computational effort. Indeed, it requires to solve the continuous relaxation LP^e for which the computing time grows up to 67.47 seconds, on average, with some rare instances were the time exceeds 2000 seconds. Lower bound LB_0 runs, on average, in at most one second, while for $LB_0(\tilde{t})$, LB_1 and LB_2^I the computing time is negligible and is not reported.

		LB_0				$LB_0(\tilde{t})$			LB_1			LB_2^I			LB_3	
class	$\|N\|$	# opt	avg gap	max gap	avg time	# opt	avg gap	max gap	# opt	avg gap	max gap	# opt	avg gap	max gap	# opt	avg time
I	89.00	144	0.04	1	0.16	135	0.10	1	13	1.35	2	27	1.03	2	150	0.49
II	147.53	149	0.01	1	0.31	112	0.31	2	8	1.84	4	15	1.55	3	150	3.55
III	164.94	150	0.00	0	0.32	90	0.49	3	0	2.45	4	2	2.16	4	150	3.14
IV	208.98	147	0.02	1	0.52	118	0.22	2	1	2.15	4	4	1.57	4	150	17.45
V	241.21	148	0.01	1	0.69	142	0.05	1	2	2.45	4	12	1.83	4	150	58.50
VI	531.69	147	0.02	1	0.70	124	0.17	1	0	2.93	5	0	2.55	4	150	42.04
VII	241.50	133	0.11	1	0.68	115	0.24	2	1	2.32	4	4	2.03	4	150	56.61
VIII	326.86	148	0.01	1	0.75	121	0.25	2	1	2.49	4	4	1.87	3	150	17.39
IX	554.91	150	0.00	0	0.69	133	0.13	2	0	2.55	4	2	2.21	3	150	23.13
X	352.15	150	0.00	0	0.83	132	0.30	3	0	2.59	5	0	2.21	4	150	30.96
		1466	**0.02**		**0.56**	**1222**	**0.23**		**26**	**2.31**		**70**	**1.90**		**1500**	**25.33**

Table 3: Performance of the Lower Bounds: instances grouped by groups

Note that although LB_3 equals the optimal solution value in all the 1500 instances we generated, this equality cannot always hold if $\mathcal{P} \neq \mathcal{NP}$. For the BPP it is known that the continuous relaxation of the exponential-size formulation almost always provides the optimal solution value. The *Integer Round-Up Property* (IRUP) states that the value of the LP relaxation of the exponential-size formulation, rounded up to the closest integer, yields the optimal solution value. However, for BPP the IRUP property does not hold (see, e.g., Caprara et al. [12]). The TBPP generalizes the BPP, so it also cannot exhibit the IRUP property.

7.2. Heuristic algorithms

We implemented three heuristic algorithms. The first one, called "Greedy" in the following, is made by running the four versions of the first fit method described in Section 5 (First-Fit-1 and First-Fit-2 applied to the original and lifted instance) and returning the best of these solutions.

The "H-Rolling" heuristic implements the rolling horizon method introduced in Section 5. The method uses a parameter Δ to define the rolling time window. We performed preliminary tests with $\Delta = \{10, 20, 30, 40\}$, on a subset of instances. On the basis of these experiments, we selected the value $\Delta = 30$ for our complete computational tests. We also made some preliminary tuning on the time limit given to `Cplex` for the solution of each restricted ILP^c model and we finally set the time limit to 10 seconds for instances with $|T| < 100$ and to 30 seconds when $|T| \geq 100$.

The last heuristic "H-Diving" is the diving method described in Section 6.3. Initially, it solves the continuous relaxation LP^{e}with the column generation method described in Section 6.1, which provides both LB_3 and the starting point of the algorithm. Then, the branch-decision-tree defined by branching rule BR-1 is partially explored using the token-rule given in Section 6.3. For the computation of the root node (LB_3) we set the time limit to 3500 CPU seconds, while 100 seconds are allowed for the tree exploration. The same 3600 seconds time limit will be used for the exact algorithms. Table 4 gives the results for the three methods. The columns report on the number of times the heuristic solution is equal to the optimal solution (# opt), the average and maximum absolute gap with respect to the optimum value (avg gap, max gap), and the average and max

		Greedy			H-Rolling					H-Diving				
			avg	max		avg	max	avg	max		avg	max	avg	max
$\|T\|$	$\|N\|$	# opt	gap	gap	# opt	gap	gap	time	time	# opt	gap	gap	time	time
10	54.90	46	0.57	2	100	0.00	0	1.87	10.00	100	0.00	0	0.46	7.02
20	88.39	36	0.84	2	92	0.08	1	4.63	10.00	100	0.00	0	1.93	15.05
30	121.43	32	1.04	3	79	0.28	3	5.53	10.00	100	0.00	0	5.92	65.20
40	154.10	30	1.10	3	70	0.34	2	5.95	10.32	99	0.02	2	12.38	130.38
50	186.70	36	1.03	3	75	0.32	3	5.81	20.00	97	0.06	3	18.93	155.72
60	219.61	34	1.06	3	75	0.29	2	6.88	20.01	91	0.15	3	32.25	202.22
70	252.50	29	1.12	3	72	0.33	2	7.32	20.08	87	0.26	3	44.49	299.81
80	285.93	30	1.13	3	73	0.30	2	7.41	20.43	76	0.47	3	59.06	332.32
90	318.86	30	1.10	4	77	0.24	2	8.70	25.77	81	0.42	4	71.37	1454.90
100	351.78	27	1.10	4	78	0.25	2	17.44	60.68	69	0.61	4	98.22	2615.81
110	385.04	27	1.14	3	79	0.26	3	19.77	65.81	63	0.69	3	84.16	580.28
120	417.73	31	1.09	4	78	0.26	2	20.06	82.45	62	0.70	4	91.98	1272.17
130	451.12	28	1.14	4	72	0.33	3	19.57	80.23	59	0.80	4	102.97	1375.88
140	483.61	31	1.09	3	74	0.31	2	19.74	78.42	50	0.90	3	127.66	1701.14
150	516.45	29	1.14	3	76	0.31	3	19.68	79.62	48	0.93	3	132.00	602.14
		476	**1.05**		**1170**	**0.26**		**11.36**		**1182**	**0.40**		**58.92**	

Table 4: Performance of the heuristic algorithms: instances grouped by time steps

computing time, when it is relevant. Each row provides results on the 100 instances we generated for each $|T|$ value. The last row summarises the above results on all the 1500 instances. Table 5 provides the same information grouped by classes (we remind that in this case we have 150 instances per row). The Greedy is extremely fast and its computing time is not reported. It is able

		Greedy			H-Rolling					H-Diving				
			avg	max		avg	max	avg	max		avg	max	avg	max
class	$\|N\|$	# opt	gap	gap	# opt	gap	gap	time	time	# opt	gap	gap	time	time
I	89.00	115	0.24	2	147	0.02	1	0.05	1.20	150	0.00	0	1.41	42.12
II	147.53	78	0.49	2	126	0.16	1	1.89	30.04	147	0.02	1	16.45	135.91
III	164.94	55	0.66	2	118	0.21	1	2.16	10.13	145	0.05	2	14.16	139.91
IV	208.98	14	1.56	3	123	0.20	2	12.26	40.77	107	0.59	3	62.22	246.13
V	241.21	13	1.85	4	74	0.65	3	20.94	79.62	102	0.77	4	111.04	2615.81
VI	531.69	28	1.06	3	128	0.17	2	14.53	61.98	83	0.67	3	100.78	595.11
VII	241.50	6	1.92	3	108	0.37	3	19.30	82.45	106	0.71	3	104.86	1701.14
VIII	326.86	44	0.93	3	120	0.23	3	13.24	60.63	115	0.38	3	50.24	428.01
IX	554.91	101	0.36	2	138	0.08	1	9.26	69.45	128	0.16	2	48.60	472.91
X	352.15	22	1.39	4	88	0.52	3	19.93	80.50	99	0.66	4	79.43	425.49
		476	**1.05**		**1170**	**0.26**		**11.36**		**1182**	**0.40**		**58.92**	

Table 5: Performance of the heuristic algorithms: instances grouped by class

to provide the optimal solution for about one third of the instances, while H-Rolling and H-Diving for about four over five solutions. H-Diving finds twelve more optimal solutions than H-Rolling, but looking at each group of instances with the same $|T|$ one can see that no one of the two algorithms dominates the other. If one looks at the single instances (not reported here) it is possible to find a sort of complementarity between the two methods: one solves instances not solved by the other and

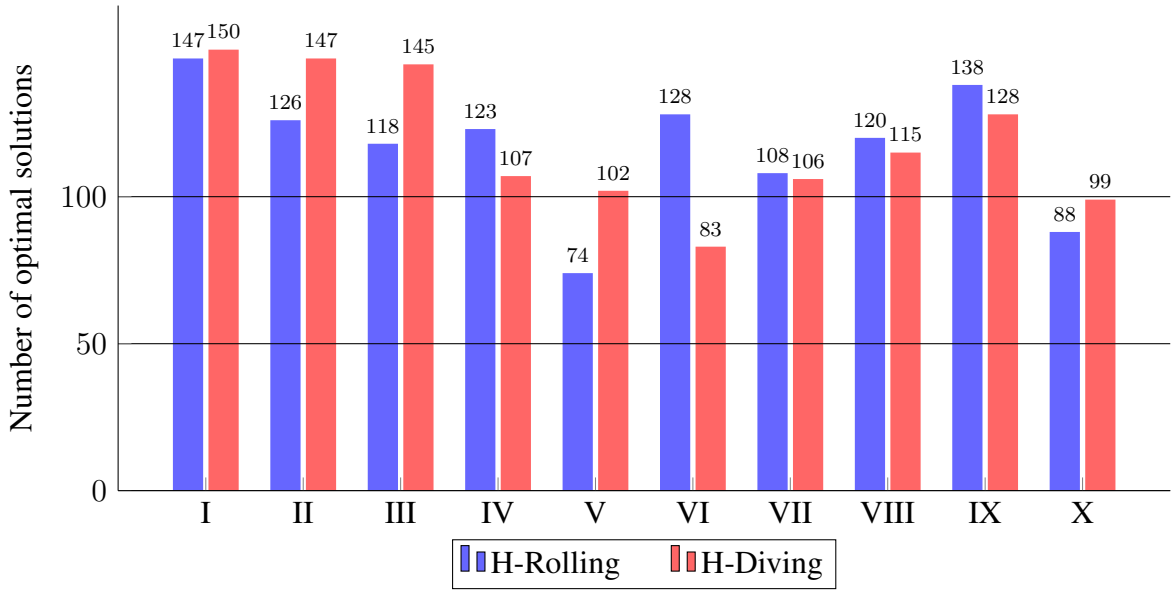


Figure 4: Heuristic performances by instance classes

vice versa. The same considerations apply to the results grouped by class, shown in Table 5. For example H-Diving improves upon H-Rolling for classes I, II, III, V and X, while H-Rolling is the winner in the other classes. H-Rolling is able to find all the 100 optimal solutions only for $|T| = 10$, then its performances slightly worsen, and for $|T| = (40, \ldots, 150)$ the number of optimal solutions found ranges between 70 and 80. H-Diving, instead, finds all the optimal solutions for $|T| \leq 30$, then its performances constantly decrease while $|T|$ increases. However, even when it finds only 48 optimal solutions ($|T| = 150$) against the 76 found by H-Rolling, there are some instances where H-Diving beats H-Rolling (and vice versa).

From Table 5 and Figure 4 one can see that class I is definitely the easiest for all algorithms, while classes II and III are also easy for H-Diving, although the computing times grows, on average, from 1.41 seconds to 16.45 seconds. Instances of classes IV and V are the most difficult to solve for H-Rolling and H-Divng, respectively.

7.3. Exact algorithms

We started the computational analysis of the exact algorithms by comparing the results of the Vector Packing Solver by Brandão and Pedroso [9] with that of the polynomial-size formulation (2)–(6), solved by `Cplex` and with the exponential-size formulation solved by our branch-and-price algorithm introduced in Section 6. Table 6 reports the results for instances with $|T| = 10, 15$. Each row refers to the 10 instances of a given class, and the columns show the results for the Vector Packing Solver 'VPSolver', for the polynomial-size model 'ILP^c' and for the exponential-size formulation 'ILP^e'. For each algorithm, we provide the number of optimal solutions found (# opt) and the average computing time over the solved instances. The column labeled '$|N|$' reports the average number of items, while the column labeled '#' reminds the number of instances tested in each class. A time limit of 600 seconds has been given to each algorithm. For $|T| = 10$,the Vector

			$\|T\| = 10$						$\|T\| = 15$					
			VPSolver		ILP^c		ILP^e		VPSolver		ILP^c		ILP^e	
class	$\|N\|$	#	# opt	time	# opt	time	# opt	time	# opt	time	# opt	time	# opt	time
I	19.0	10	10	0.01	10	0.02	10	0.02	10	0.03	10	0.03	10	0.04
II	30.2	10	10	0.05	10	0.06	10	0.15	10	0.33	10	0.21	10	0.22
III	36.9	10	10	0.05	10	0.14	10	0.04	10	0.46	10	0.69	10	0.09
IV	46.1	10	10	0.33	10	0.45	10	0.71	10	3.66	10	1.98	10	1.10
V	53.8	10	10	0.48	10	1.93	10	0.47	10	9.70	10	6.20	10	1.59
VI	88.5	10	3	176.91	10	3.91	10	1.85	0	M.L.	10	48.79	10	6.09
VII	53.4	10	10	0.34	10	0.47	10	0.45	10	6.58	10	8.06	10	1.36
VIII	64.4	10	10	18.52	10	0.91	10	0.55	8	108.29	9	4.06	10	1.49
IX	87.6	10	4	193.75	10	5.04	10	1.13	0	M.L.	10	19.69	10	7.03
X	69.1	10	10	4.40	10	125.17	10	0.26	5	124.09	10	20.14	10	1.51
			87	**39.48**	**100**	**13.81**	**100**	**0.56**	**73**	**31.64**	**99**	**10.98**	**100**	**2.05**

Table 6: Performance comparison of exact algorithms on small instances (time limit 600 secs)

Packing approach is not able to solve seven and six instances from classes VI and IX, respectively. For $|T| = 15$, it is able to solve only 73 over 100 instances. In this case, the unsolved instances are due to the excessive memory usage (entry 'M.L.' in the table). The computing time drastically increases for some instances. This is probably due to the fact that VPSolver is intended to solve a more general problem than the TBPP. The ILP^c model is able to solve all instances with $|T| = 15$ but one, using 31.64 CPU seconds on average. The ILP^e solves all instances in short times (on average 2.05 seconds) and is two order of magnitude faster than the VPSolver. On the basis of these results, we decided that using VPSolver to solve the problem is not viable due to memory limitation ad high computing times, so we disregard this approach for the next experiments.

We then performed computational experiments to assess the effectiveness of ILP^c and of our overall algorithm B&P$^+$ to solve the TBPP. We set a time limit of 3600 seconds for each algorithm and instance. In Tables 7 and 8 we report, for ILP^c and B&P$^+$, the number of optimal solution and the average and maximum absolute gap and computing time. For B&P$^+$ we additionally report the average root time, the average number of columns in the root LP, and the average number of nodes explored. The average and maximum values are computed on the instances which are solved to a proven optimum. The two tables group the instances by $|T|$ and class, respectively.

From Table 7 one can see that B&P$^+$ dominates ILP^c for all $|T| < 150$, since it finds always more optimal solutions in a shorter average time. The ILP^c reaches the time limit for $|T| \geq 20$, while B&P$^+$ for $|T| \geq 90$. For $|T| = 150$ B&P$^+$ finds two less optima than ILP^c in a comparable running time. The analysis by instance classes in Table 8 confirms that B&P$^+$ dominates ILP^c class by class, both by number of optimal solutions found and computing time. The first three classes appear to be easy for both algorithm, while class IV is also easy, but ILP^c fails on three instances. Class X remains the most difficult for both methods.

In order to give a graphical representation of the relative performance of the two exact algorithms, we report a performance profile in Figure 5. For each instance, we compute a normalized time τ as the ratio of the computing time of the considered configuration over the minimum computing time for solving the instance to optimality. For each value of τ the vertical axis reports the percentage of the instances for which the corresponding configuration spent at most τ times the computing time

of the fastest configuration. The curves start from the percentage of instances in which the corresponding configuration is the fastest and at the right end of the chart, we can read the percentage of instances solved by a specific algorithm. The best performance are graphically represented by the curves in the upper part of Figure 5. B&P$^+$ is the fastest one for approximatively 70% of the instances and it is able to solve to problem optimality 98% of them. On the other hand, ILP^{c}is able to solve only 90% of the instances within the same time limit of 1 hour, and only 65% and 85% of the instances by allowing 10 and 100 times more time than the one required by B&P$^+$, respectively. Figure 5 graphically demonstrates that B&P$^+$ compares favorably to ILP^c on the testbed of the 1500 considered instances.

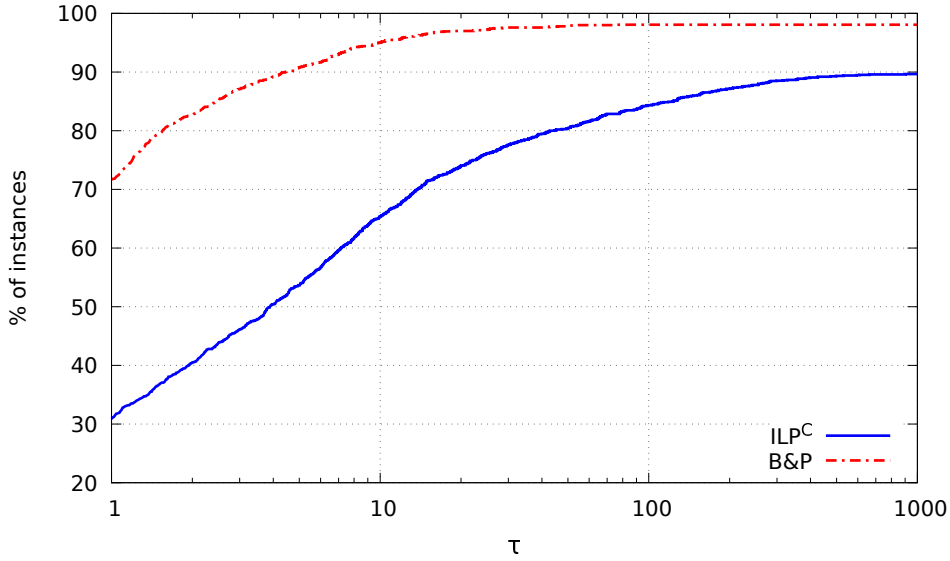


Figure 5: Performance profile of the exact methods

We complete our analysis by studying the contribution of each component of the overall algorithm B&P$^+$ to the solution of the TBPP, see Tables 9 and 10. The first group of three columns (labeled H-Rolling+LB_0) refers to the first phase of the algorithms where we compute all the lower bounds, but LB_3 and we run the Greedy and the H-Rolling heuristic. In the columns we report the number of instance solved (# opt) and the average and maximum running times, computed with respect to the solved instances. The second group of three columns (labeled H-Rolling+LB_3) refers to the instances that are not solved in the previous phase and shows the results obtained when lower bound LB_3 (i.e., the LP relaxation of the root node of the branch-decision-tree) is computed. Again we report the results evaluated only for the solved instances. Note that in column labeled "#opt" we report the total number of instances solved in the first two phases of the algorithm, and in brackets the number of new optima. The next group of three columns (labeled H-Diving) shows the results on the remaining unsolved instances, after the execution of the diving search. The last group of five columns gives the results of the application of the B&P$^+$ search to the instances that are not solved by the heuristics and lower bounds.

8. Conclusions

In this paper, we studied the Temporal Bin Packing Problem (TBPP), a challenging generalization of the classical Bin Packing Problem where each item consumes the bin capacity during a given time window. The goal is to determine the minimum number of bins to pack all the items while, at the same time, respecting the bin capacity at any instant of time. We have proposed and studied the first two mathematical formulations for the TBPP, the first one with a polynomial number of variables and constraints and the second one with an exponential number of variables. We have introduced several upper and lower bounds for the TBPP and we have designed an exact algorithm which combines them in an effective way. Our new branch-and-price algorithm, based on column generation, is able to solve to proven optimality instances with up to 500 items and 150 time steps, in reasonable computing times.

Several are the potential future lines of research. In the recent literature, effective pseudo-polynomial size formulations have been proposed for the Bin Packing Problem. It would be interesting to study if these formulations, especially the ones based on the arc flow mechanism (see, e.g., Valério de Carvalho [39] and [20]), could be effectively used to tackle the TBPP as well. Finally, it would be interesting to introduce additional real-world features to the TBPP; for instance, precedence constraints between the items (see, e.g., Dell'Amico et al. [18]) or item class set-up costs (see, e.g., Furini et al. [25]).

9. Acknowledgments

Research supported by MIUR-Italy (Grant PRIN 2015, Nonlinear and Combinatorial Aspects of Complex Networks) and UNIMORE (Grant FAR 2018, Analysis and optimization of healthcare and pharmaceutical logistic processes).

		ILP^c						B&P$^+$								
													root			
$\|T\|$	$\|N\|$	# opt	avg gap	max gap	avg time	max time	avg nodes	# opt	avg gap	max gap	avg time	max time	avg time	max time	col	avg nodes
10	54.90	100	0.00	0.00	13.81	1234.59	662.6	100	0.00	0	1.95	10.19	0.24	3.08	103.1	7.9
20	88.39	99	0.01	1.00	100.46	t.l.	2361.5	100	0.00	0	5.24	25.19	0.90	6.06	264.7	24.1
30	121.43	96	0.04	1.00	273.49	t.l.	2476.8	100	0.00	0	9.33	75.46	2.23	16.25	524.9	48.8
40	154.10	92	0.08	1.40	396.91	t.l.	2683.6	100	0.00	0	15.74	385.20	4.23	33.68	844.7	71.2
50	186.70	93	0.08	1.39	414.36	t.l.	2683.2	100	0.00	0	27.89	935.86	5.51	48.47	1045.6	85.9
60	219.61	85	0.17	2.01	596.10	t.l.	2289.5	100	0.00	0	63.94	2168.50	9.80	88.62	1585.2	114.9
70	252.50	84	0.18	1.62	694.06	t.l.	2648.7	100	0.00	0	115.12	2622.69	14.41	173.10	2193.8	148.1
80	285.93	85	0.20	2.61	635.66	t.l.	2627.2	100	0.00	0	132.60	2964.84	18.07	201.83	2708.8	174.8
90	318.86	88	0.15	2.52	578.75	t.l.	2955.5	99	0.01	1	151.82	t.l.	30.14	1295.94	3178.4	205.6
100	351.78	86	0.16	2.00	642.66	t.l.	2528.0	99	0.01	1	168.85	t.l.	44.61	2245.15	3496.2	207.8
110	385.04	88	0.14	1.96	609.58	t.l.	2754.6	98	0.04	3	218.01	t.l.	29.79	472.09	4154.4	229.4
120	417.73	91	0.12	2.07	523.89	t.l.	2353.4	99	0.02	2	239.87	t.l.	38.23	1164.56	4588.4	239.3
130	451.12	92	0.10	2.02	631.67	t.l.	2156.0	94	0.09	3	433.91	t.l.	48.40	1253.87	4686.6	238.3
140	483.61	89	0.14	2.00	634.70	t.l.	2687.4	92	0.11	2	475.08	t.l.	65.85	1555.42	6764.3	285.7
150	516.45	92	0.09	2.34	613.20	t.l.	2605.0	90	0.15	3	560.40	t.l.	67.47	502.21	7519.6	290.1
		1360	**0.11**				**2431.5**	**1471**	**0.03**		**174.75**		**25.33**		**2396.4**	**142.3**

Table 7: Performance of the exact algorithms, instances grouped by time steps

		ILP^c						B&P$^+$								
													root			
class	$\|N\|$	# opt	avg gap	max gap	avg time	max time	avg nodes	# opt	avg gap	max gap	avg time	max time	avg time	max time	col	avg nodes
I	19.0	150	0.00	0.00	1.23	157.46	80.8	150	0.00	0	0.55	30.69	0.49	6.81	193.7	11.2
II	30.2	150	0.00	0.00	3.90	82.68	107.3	150	0.00	0	9.35	222.46	3.55	34.21	809.9	47.3
III	36.9	150	0.00	0.00	22.89	739.07	359.2	150	0.00	0	8.10	279.36	3.14	37.83	852.2	65.7
IV	46.1	147	0.02	1.00	206.61	t.l.	1444.9	150	0.00	0	60.10	1135.04	17.45	141.83	2222.9	134.7
V	53.8	128	0.19	2.61	882.97	t.l.	3449.9	146	0.04	2	396.64	t.l.	58.50	2245.15	2943.1	164.1
VI	88.5	142	0.06	1.40	359.66	t.l.	2066.4	146	0.03	1	214.72	t.l.	42.04	473.19	5238.6	289.1
VII	53.4	124	0.19	2.34	874.38	t.l.	2864.5	147	0.04	3	249.37	t.l.	56.61	1555.42	2499.9	160.2
VIII	64.4	132	0.15	1.89	586.40	t.l.	3256.9	144	0.07	3	223.86	t.l.	17.39	326.13	2770.3	178.8
IX	87.6	124	0.20	2.00	845.39	t.l.	4794.7	147	0.02	1	82.28	t.l.	23.13	368.59	2574.4	132.7
X	69.1	113	0.30	2.07	1122.77	t.l.	5890.7	141	0.09	3	501.58	t.l.	30.96	326.02	3699.5	229.7
		1360	**0.11**				**2431.5**	**1471**	**0.03**		**174.75**		**25.33**		**2396.4**	**142.3**

Table 8: Performance of the exact algorithms, instances grouped by classes

		H-Rolling+LB_0			H-Rolling+LB_3			H-Diving			B&P$^+$ search				
$\|T\|$	$\|N\|$	# opt	avg time	max time	# opt	avg time	max time	# opt	avg time	max time	# opt	avg gap	max gap	avg time	max time
10	54.90	97	1.80	10.00	100(3)	4.44	10.26	-	-	-	-	-	-	-	-
20	88.39	88	3.90	10.00	92 (4)	12.35	13.72	100 (8)	14.59	24.39	-	-	-	-	-
30	121.43	74	4.08	10.00	79 (5)	14.37	22.10	100 (21)	25.01	72.27	-	-	-	-	-
40	154.10	67	4.49	10.32	70 (3)	18.51	31.74	99 (29)	26.97	93.46	100 (1)	0.00	0.00	385.20	385.20
50	186.70	73	4.75	20.00	75 (2)	21.04	41.17	97 (22)	34.33	117.04	100 (3)	0.00	0.00	531.95	935.86
60	219.61	72	6.10	20.01	75 (3)	20.55	58.11	95 (20)	41.60	106.51	100 (5)	0.00	0.00	999.49	2168.50
70	252.50	70	6.45	20.08	72 (2)	56.69	73.83	94 (22)	38.76	98.22	100 (6)	0.00	0.00	1671.65	2622.69
80	285.93	72	6.48	20.43	73 (1)	55.79	55.79	91 (18)	44.97	100.40	100 (9)	0.00	0.00	1318.28	2964.84
90	318.86	76	8.02	25.77	77 (1)	76.80	76.80	92 (15)	51.87	116.66	99 (7)	0.13	1.00	1708.19	t.l.
100	351.78	78	14.77	60.68	78 (-)	-	-	90 (12)	66.79	141.48	99 (9)	0.10	1.00	1486.10	t.l.
110	385.04	78	17.82	65.81	79 (1)	93.17	93.17	88 (9)	62.04	126.55	98 (10)	0.33	3.00	1641.51	t.l.
120	417.73	78	19.28	82.45	78 (-)	-	-	87 (9)	57.14	153.56	99 (12)	0.15	2.00	1684.90	t.l.
130	451.12	72	17.70	80.23	72 (-)	-	-	83 (11)	61.97	123.74	94 (11)	0.53	3.00	2433.47	t.l.
140	483.61	74	18.34	78.42	74 (-)	-	-	80 (6)	52.06	98.00	92 (12)	0.55	2.00	2288.29	t.l.
150	516.45	75	16.67	64.10	76 (1)	100.67	100.67	82 (6)	64.36	114.64	90 (8)	0.83	3.00	3012.76	t.l.
		1144	**9.93**		**1170 (26)**	**28.22**		**1378 (208)**	**41.85**		**1471 (93)**	**0.35**		**1970.21**	

Table 9: Contribution of the various methods used in B&P$^+$: instances grouped by time steps

		H-Rolling+LB_0			H-Rolling+LB_3			H-Diving			B&P$^+$ search				
			avg	max		avg	max		avg	max		avg	max	avg	max
$\|T\|$	$\|N\|$	# opt	time	time	# opt	time	time	# opt	time	time	# opt	gap	gap	time	time
I	19.0	141	0.02	1.20	14 (6)	1.19	2.43	150 (3)	16.44	30.21	-	-	-	-	-
II	30.2	125	2.09	30.04	126 (1)	2.77	2.77	149 (23)	37.58	93.76	150 (1)	0.00	0.00	222.46	222.46
III	36.9	118	2.26	10.13	118 (-)	-	-	149 (31)	20.02	66.87	150 (1)	0.00	0.00	279.36	279.36
IV	46.1	121	12.32	40.77	123 (2)	15.80	17.87	141 (18)	46.96	112.07	150 (9)	0.00	0.00	729.38	1135.04
V	53.8	72	18.41	61.21	74 (2)	55.46	100.67	118 (44)	51.63	153.56	146 (28)	0.19	2.00	1739.71	t.l.
VI	88.5	125	14.76	61.98	128 (3)	57.38	76.80	136 (8)	53.52	116.99	146 (10)	0.29	1.00	2118.85	t.l.
VII	53.4	98	19.67	82.45	108 (10)	38.43	93.17	130 (22)	57.01	131.99	147 (17)	0.30	3.00	1687.26	t.l.
VIII	64.4	118	11.74	41.10	120 (2)	12.37	13.59	138 (18)	29.58	98.22	144 (6)	0.83	3.00	2627.64	t.l.
IX	87.6	138	8.12	69.45	138 (-)	-	-	147 (9)	35.70	72.27	147 (-)	1.00	1.00	t.l.	t.l.
X	69.1	88	19.73	80.50	88 (-)	-	-	120 (32)	47.44	140.60	141 (21)	0.47	3.00	2396.26	t.l.
		1144	**9.93**		**1170 (26)**	**28.22**		**1378 (208)**	**41.85**		**1471 (93)**	**0.35**		**1970.21**	

Table 10: Contribution of the various methods used in B&P$^+$: instances grouped by classes

References

References

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

[2] C. Alves, J. V. de Carvalho, F. Clautiaux, and J. Rietz. Multidimensional dual-feasible functions and fast lower bounds for the vector packing problem. *European Journal of Operational Research*, 233(1):43–63, 2014.

[3] E. Angelelli and C. Filippi. On the complexity of interval scheduling with a resource constraint. *Theoretical Computer Science*, 412(29):2650,2657, 2011.

[4] E. Angelelli, N. Bianchessi, and C. Filippi. Optimal interval scheduling with a resource constraint. *Computers & Operations Research*, 51(3):268–281, 2016.

[5] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46 (3):316–329, 1998.

[6] M. Bartlett, A. Frisch, Y. Hamadi, I. Miguel, S. Tarim, and C. Unsworth. The temporal knapsack problem and its solution. In R. Bartak and M. Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2005)*, Lecture Notes in Computer Science, vol 3524, pages 34–48. Springer, Berlin, Heidelberg, 2005.

[7] G. Belov and H. Rohling. LP bounds in an interval-graph algorithm for orthogonal-packing feasibility. *Operations Research*, 61:483–497, 2013.

[8] H. Ben Amor, J. Desrosiers, and J. Valério de Carvalho. Dual-optimal inequalities for stabilized column generation. *Operations Research*, 54(3):454–463, 2006.

[9] F. Brandão and J. P. Pedroso. Bin packing and related problems: General arc-flow formulation with graph compression. *Computers & Operations Research*, 69:56–67, 2016.

[10] A. Caprara and P. Toth. Lower bounds and algorithms for the 2-dimensional vector packing problem. *Discrete Applied Mathematics*, 111(3):231–262, 2001.

[11] A. Caprara, F. Furini, and E. Malaguti. Uncommon dantzig-wolfe reformulation for the temporal knapsack problem. *INFORMS Journal on Computing*, 25(3):560–571, 2013.

[12] A. Caprara, M. Dell'Amico, J. Díaz Díaz, M. Iori, and R. Rizzi. Friendly bin packing instances without integer round-up property. *Mathematical Programming-B*, 150:5–17, 2015.

[13] A. Caprara, F. Furini, E. Malaguti, and E. Traversi. Solving the temporal knapsack problem via recursive dantzig-wolfe reformulation. *Information Processing Letters*, 116(5):379–386, 2016.

[14] F. Clautiaux, S. Hanafi, R. Macedo, M.-A. Voge, and C. Alves. Iterative aggregation and disaggregation algorithm for pseudo-polynomial network flow models with side constraints. *European Journal of Operational Research*, 258(2):467–477, 2017.

[15] E. Coffman Jr. and J. Csirik. A classification scheme for bin packing theory. *Acta Cybernetica*, 18(1):47–60, 2007.

[16] E. G. Coffman Jr., J. Csirik, G. Galambos, S. Martello, and D. Vigo. Bin packing approximation algorithms: Survey and classification. In P. M. Pardalos, D. Du, and R. L. Graham, editors, *Handbook of Combinatorial Optimization*, pages 455–531. Springer New York, 2013. doi: 10.1007/978-1-4419-7997-1_35.

[17] J.-F. Côté, M. Dell'Amico, and M. Iori. Combinatorial Benders' cuts for the strip packing problem. *Operations Research*, 62(3):643–661, 2014.

[18] M. Dell'Amico, J. Diaz-Diaz, and M. Iori. The bin packing problem with precedence constraints. *Operations Research*, 60(6):1491–1504, 2012.

[19] M. Dell'Amico, M. Delorme, M. Iori, and S. Martello. Mathematical models and decomposition methods for the multiple knapsack problem. *European Journal of Operational Research*, 274(3):886–899, 2019.

[20] M. Delorme and M. Iori. Enhanced pseudo-polynomial formulations for bin packing and cutting stock problems. *Accepted for publication on INFORMS Journal on Computing*, 2018.

[21] M. Delorme, M. Iori, and S. Martello. Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research*, 255(1):1–20, 2016.

[22] M. Delorme, M. Iori, and S. Martello. Logic based Benders' decomposition for orthogonal stock cutting problems. *Computers & Operations Research*, 78:290–298, 2017.

[23] M. Delorme, M. Iori, and S. Martello. BPPLIB: a library for bin packing and cutting stock problems. *Optimization Letters*, 12(2):235–250, 2018.

[24] G. Desaulniers, J. Desrosiers, and M. Solomon, editors. *Column generation*, volume 5. Springer Science & Business Media, 2006.

[25] F. Furini, M. Monaci, and E. Traversi. Exact approaches for the knapsack problem with setups. *Computers & OR*, 90:208–220, 2018.

[26] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859, 1961.

[27] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting stock problem - part II. *Operations Research*, 11(6):863–888, 1963.

[28] T. Gschwind and S. Irnich. Stabilized column generation for the temporal knapsack problem using dual-optimal inequalities. *OR Spectrum*, 39(2):541–556, 2017.

[29] R. Haessler and P. Sweeney. Cutting stock problems and solution procedures. *European Journal of Operational Research*, 54(2):141–150, 1991.

[30] K. Hessler, T. Gschwind, and S. Irnich. Stabilized branch-and-price algorithms for vector packing problems. *European Journal of Operational Research*, 271(2):401–419, 2018.

[31] D. Johnson. *Near-optimal bin-packing algorithms*. PhD thesis, MIT, 1985.

[32] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[33] A. Lodi, S. Martello, and D. Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123(1):379–396, 2002.

[34] A. Lodi, S. Martello, M. Monaci, and D. Vigo. *Two-Dimensional Bin Packing Problems*, pages 107–129. John Wiley & Sons, Ltd, 2013.

[35] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, New York, 1990.

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

[37] D. Pisinger and M. Sigurd. Using decomposition techniques and constraint programming for solving the two-dimensional bin packing problem. *INFORMS Journal on Computing*, 19(1): 36 – 51, 2007.

[38] M. Serairi and M. Haouari. A theoretical and experimental study of fast lower bounds for the two-dimensional bin packing problem. *RAIRO-Operations Research*, 52(2):391–414, 2018.

[39] J. Valério de Carvalho. LP models for bin packing and cutting stock problems. *European Journal of Operational Research*, 141(2):253–273, 2002.

[40] J. Valério de Carvalho. Using extra dual cuts to accelerate column generation. *INFORMS Journal on Computing*, 17(2):175–182, 2005.

[41] F. Vanderbeck. Branching in branch-and-price: a generic scheme. *Mathematical Programming*, 130(2):249–294, 2011. doi: 10.1007/s10107-009-0334-1.

[42] G. Wäscher, H. Haußner, and H. Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130, 2007.

[43] L. Wei, Z. Luo, R. Baldacci, and A. Lim. A new branch-and-price-and-cut algorithm for one-dimensional bin packing problems. *Accepted for publication on INFORMS Journal on Computing*, 2018.