Three-Dimensional Bin Packing and Mixed-Case Palletization

Samir Elhedhli, Fatma Gzara, Burak Yildiz

Please scroll down for article—it is on subsequent pages

With 12,500 members from nearly 90 countries, INFORMS is the largest international association of operations research (O.R.) and analytics professionals and students. INFORMS provides unique networking and learning opportunities for individual professionals, and organizations of all types and sizes, to better understand and use O.R. and analytics tools and methods to transform strategic visions and achieve better outcomes.
For more information on INFORMS, its publications, membership, or meetings visit http://www.informs.org

Three-Dimensional Bin Packing and Mixed-Case Palletization

Samir Elhedhli,[a] Fatma Gzara,[a] Burak Yildiz[a]

[a] Department of Management Sciences, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada
Contact: elhedhli@uwaterloo.ca, http://orcid.org/0000-0003-1922-8286 (SE); fgzara@uwaterloo.ca (FG);
byildiz@uwaterloo.ca, http://orcid.org/0000-0001-5622-874X (BY)

Abstract. Despite its wide range of applications, the three-dimensional bin-packing problem is still one of the most difficult optimization problems to solve. Currently, medium- to large-size instances are only solved heuristically and remain out of reach of exact methods. This is particularly true for its practical variant, the mixed-case palletization problem, where item support is needed. This and the lack of a realistic benchmark data set are identified as major research gaps by a recent survey. In this work, we propose a novel formulation and a column-generation solution approach, where the pricing subproblem is a two-dimensional layer-generation problem. Layers are highly desirable in practical packings as they are easily packable and can accommodate important practical constraints such as item support, family groupings, isle friendliness, and load bearing. Being key to the success of the column-generation approach, the pricing subproblem is solved optimally as well as heuristically and is enhanced by using item grouping, item replacement, layer re-organization, and layer spacing. We conduct extensive computational experiments and compare against existing approaches. We also use industrial data to train and propose a realistic data set. The proposed approach outperforms the best-performing algorithm in the literature on most instances and succeeds to solve practical size instances in very reasonable computational times.

1. Introduction

Three dimensional packing problems are defined on a set of rectangular boxes/items with given dimensions to be packed in one or more bins. According to Martello et al. (2000), they come in three variants, depending on the characteristics of the bin and the objective: knapsack loading, container loading, and bin packing. The first seeks to pack a subset of the given items in one or more bins such that the total value of the packed items is maximized; the second packs items in a single bin with fixed width and depth and unlimited height such that the height position of the top-most item is minimized; and the third packs a set of items in an infinite set of fixed-dimension bins with the objective of minimizing the number of bins used.

In this work, we focus on the two last versions related to three-dimensional packing problems, namely, the three-dimensional strip-packing problem (3DSPP) and the three-dimensional bin-packing problem (3DBPP). In 3DBPP, the goal is to find the minimum number of rectangular bins to orthogonally pack a set of rectangular items, whereas in the 3DSPP, the goal is to find a packing of all items inside an infinite-height bin such that the height of the bin is minimized. All packing problems require that items do not overlap and are placed perpendicular to the edges of the bin. We do not allow item rotations or impose additional constraints such as item support or the so-called guillotine constraints. The 3DSPP is of particular importance in this work. We use it as a basis for a column-generation framework with the aim of generating promising layers efficiently to build solutions for 3DBPP.

The mixed-case palletization problem (MCPP) is an extension of the 3DBPP that incorporates practical features, such as bin stability and item support. This problem arises in logistics and warehouse operations. Pallets and containers, being the most common platform for shipping, are essentially three-dimensional packings with side constraints. Our interest in MCPP is inspired by an industry project for a global warehousing and logistics company. Building optimized three-dimensional pallets is a major bottleneck in its highly automated warehouse-sorting, retrieval, and palletization systems. A modern automated warehouse is expected to pack thousands of items into hundreds of industry-sized pallets daily. Owing to the dynamic and

fast-paced nature of the packaging operation, the warehousing and logistics company seeks to plan a pallet in less than two minutes.

The 3DBPP, and by extension the MCPP, remains one of the most difficult mixed-integer optimization problems to solve. Although a direct extension of the two-dimensional bin-packing problem, the third dimension introduces substantial complications, such as vertical support, bin stability, and load bearing (Bortfeldt and Wäscher 2013). Currently, medium- to large-size 3DBPP instances are only solved heuristically and remain out of reach of exact methods. Moreover, methodologies that account for practical features cannot solve even the smallest of industry instances in reasonable times. This research is highly motivated by the lack of fast and scalable solution methodologies for the 3DBPP and MCPP. Moreover, a recent survey by Zhao et al. (2016), focusing on solution methodologies and their comparative performance on benchmark data sets, identified three main research gaps: the lack of approaches for multiple container-size problems, the inadequate handling of real-world practical constraints such as bin stability and vertical support, and the absence of realistic benchmark data sets. This work addresses the last two. We provide an approach that tackles bin stability through the concept of layers and provide a realistic benchmark data set based on industrial data. The use of layers has numerous advantages related to stability and support. It also enables the use of a column-generation approach where the subproblem generates two-dimensional packings.

The main contributions of this work are threefold. First, we propose a novel formulation and a column-generation framework based on layers. We explicitly model layers and devise branch-and-price and column-generation methods. Being key to the success of the approach, we focus on the layering subproblem, solving it both exactly to achieve tight lower bounds and heuristically for quick warm-starting. To construct feasible solutions, we propose strategies for layer selection and bin construction, as well as a simple placement heuristic to pack remaining items. Second, we address one of the most important practical constraints, which is item support, and highlight its relationship with the proposed layering approach and the construction heuristic we adopt. Increasing the density of layers implicitly provides bins with high stability and vertical support (Zhao et al. 2016). Therefore, we maximize layer density through item groupings that we call superitems and prioritize denser layers when bins are constructed. We also propose an optimization model to evenly distribute items in layers for better spacing and increased support. Third, we analyze a large set of industrial data and propose a framework to generate realistic instances for 3DBPP that we hope will be used as a basis for future benchmarking. The analysis identifies item types based on their dimensional proportions, volumes, and frequency of occurrence and uses these features to construct statistical distributions and parameters that are utilized to generate instances that resemble real-life palletization problems. To the best of our knowledge, this is the first work to propose a rigorous layer-based column-generation approach for 3DBPP that is capable of handling industry-size instances, addresses practical constraints, and outperforms previous approaches. The results show a clear superiority in terms of optimal number of bins used and solution times as well as bin stability and vertical support.

The rest of the paper is organized as follows. Section 2 reviews the current literature with a focus on placement methods as well as exact and heuristic approaches. Section 3 discusses the layer definition and mathematical formulations, and Section 4 details the solution approach using column generation. Section 5 provides layer- and bin-improvement strategies and describes the item-spacing approach. Section 6 discusses practical requirements. Section 7 is devoted to numerical testing. It presents the realistic instance generator, compares to state-of-the-art approaches, and proposes a preliminary approach that explicitly tackles vertical support. Section 8 concludes the work.

2. Literature Review

Although the 3DBPP may be modeled and solved as a mixed-integer program (MIP), exact solution methodologies are scarce. Instead, the literature offers a variety of heuristics. In what follows, we review both approaches but focus on the best-performing heuristic and exact methods, bounding schemes, and commonly used benchmark data sets. For recent reviews on 3DBPP and container-loading problems, we refer the reader to Zhao et al. (2016) and Bortfeldt and Wäscher (2013).

2.1. Heuristic Methods

The 3DBPP literature includes several well-performing heuristic approaches that are capable of solving larger and more realistic instances compared with exact methodologies. They are generally of two types: placement-point methods and metaheuristics. Martello et al. (2000) were the first to introduce a placement-point method for the two-dimensional bin-packing problem (2DBPP) and 3DBPP based on corner points. A corner point is the result of the intersection between the three planes formed by the right, back, and top sides of items already

placed in a bin. The bin is divided in two parts using a staircase that separates already-placed items from the rest of the available space. They also introduced benchmark instances that were used in most subsequent work. Crainic et al. (2008) extended this approach by introducing the concept of extreme points. The latter result from the projections of the right, back, and top of items already placed in a bin. This method does not use an envelope to separate placed items from the rest of the bin as in the corner-point approach. Using projections increases the number of candidate points for placement. Figure 1, (a) and (b), shows a 2D representation of the corner point and the extreme point methods, respectively. Zhu et al. (2012b) presented a space-defragmentation placement approach, where dead space between placed items is consolidated by pushing the items to the edges. New items are then placed in the resulting space, and the existing items are pushed back.

Based on the extreme point method, Crainic et al. (2008) provided first-fit decreasing and best-fit decreasing heuristics, both of which use item-sorting rules based on height–volume, volume–height, and so forth. After the items are sorted, the first-fit decreasing heuristic places the next item in the first open bin with a feasible extreme point placement. If there are no feasible placements, a new bin is opened. The best-fit decreasing heuristic, on the other hand, evaluates extreme point item pairs based on a merit function and places them at the best point available. Zhu and Lim (2012) proposed a greedy look-ahead tree search algorithm. It starts by generating blocks of items and places them at the corner of the bin. The residual space created provides new corners that are, in turn, used to place new blocks. At each node of the search tree, they select a limited number of best placements and create child nodes by placing a new block at each. Zhu et al. (2012b) presented a construction heuristic based on the extreme-point and space-defragmentation methods. Items are placed at extreme points, and space is consolidated by pushing items out to the edges. They further improve the solution by moving items from a bin to another using space defragmentation. Faroe et al. (2003) proposed a guided local search heuristic, where the neighborhood is defined by either moving an item in any orthogonal direction or moving it to the same location in another bin while allowing overlapping. To find a feasible placement of items for a bin, they minimize the overlap between items and guide the solution by penalizing consecutive overlaps.

Two Tabu Search heuristics were proposed for the problem. The first is due to Lodi et al. (2002). It starts by packing one item per bin. The neighborhood is defined by picking one bin and placing all its items into other bins. Items are placed in layers in two steps. Height-first and area-first strategies are used in steps 1 and 2, respectively. At the end of each step, a one-dimensional bin-packing problem (1DBPP) is solved to pack the layers into the bins. The second Tabu Search heuristic is TS^2PACK due to Crainic et al. (2009), which also has two main steps. The first step determines the set of items to be packed into each bin, while the second step determines the position of each item in the bins. They use the extreme point-based first-fit descending heuristic, which they introduced for 3DBPP, to find an initial solution. In the first step, the neighborhood is constructed by either swapping items from different bins or moving an item to another bin, where the bin dimensions are relaxed. In the second step, they find a feasible packing in a bin by swapping the relative positions of items.

Parreño et al. (2010) presented a hybrid heuristic that combines variable neighborhood descent and the greedy randomized adaptive search procedure (GRASP). It starts by constructing an initial solution using the

Figure 1. 2D Representation of Placement Methods

Corner point   Extreme point
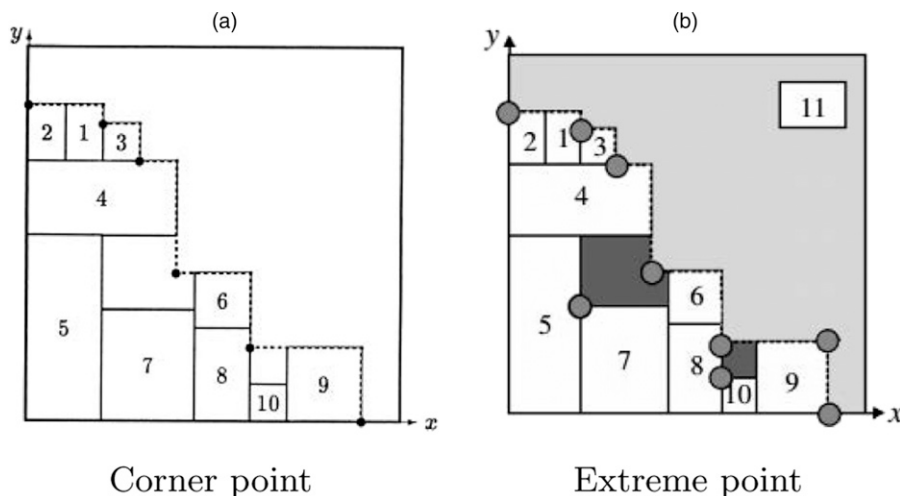
maximal space algorithm for the container-loading problem that is based on GRASP. It then uses four improvement procedures in a variable neighborhood descent scheme by testing a move in each neighborhood. Wu et al. (2010) presented a genetic algorithm based on varying the relative positions of items within a MIP. The chromosomes represent the order of items to be packed and their orientations, which are initially assigned by fixing an item sequence and randomly determining orientations. They use single point cross-over to keep the order of the items relatively stable, a sequential and a random repair scheme to fix the chromosomes, and two different mutation schemes.

Toffolo et al. (2017) introduced a heuristic decomposition method to solve the multiple container-size loading problem introduced by the Renault Challenge (Clautiaux et al. 2015). Taking advantage of the structure included in the challenge, they first form stacks and then place them to generate bins. The stacks are generated by solving a two-dimensional packing problem. The idea is similar to the current work, but is neither based on column generation nor provides solution guarantees. Zhu et al. (2012a) proposed a column-generation approach where each column is a certain arrangement of items in a single bin. The columns are generated by solving a single container-loading subproblem. Because the subproblems are difficult to solve, they solve a single-container knapsack problem to generate approximate columns.

Out of all these heuristics, the space-defragmentation approach of Zhu et al. (2012b) is found to perform the best.

2.2. Exact Methods

Because of the complexity of the 3DBPP, the literature offers few exact solution methods. Chen et al. (1995) provided the first MIP formulation for 3DBPP based on the relative positions of items, which we refer to as the relative positioning model (RPM) in the rest of this paper. Their work is later extended by Wu et al. (2010) to allow for item orientations and by Junqueira et al. (2012) and Paquay et al. (2016) to accommodate practical constraints. Hifi et al. (2010) provided several lower bounds to RPM to decrease the solution time. They are based on the linear programming (LP) relaxation and valid inequalities. Junqueira et al. (2012) accounted for cargo stability and load bearing. They provided a framework to generate groups of items, which we expand on in this paper. Paquay et al. (2016) extended RPM by accounting for load bearing, item orientations, container shapes, weight distribution, and stability. Both Junqueira et al. (2012) and Paquay et al. (2016) solve their models using a commercial solver.

Martello et al. (2000) presented an enumerative two-step tree-search algorithm based on the corner-point placement method. Their work is extended by Martello et al. (2007), who provided an improved solution algorithm called "algorithm 864." This algorithm determines which items are to be packed into which bin at each node of the tree, ignoring placement. The feasibility of the formed bins is verified by using constraint programming. Additionally, they proposed three lower bounds for 3DBPP. The first, named L_0, is the continuous lower bound, calculated by dividing the total volume of the items by the volume of a bin. The second, named L_1, is calculated by determining large items that can only be placed one behind the other (e.g., $w_i > W/2$ and $d_i > D/2$) and placing them by using 1DBPP. The third, named L_2, is calculated by adding a continuous lower bound to L_1 for the smaller items. A similar two-step approach was provided by Fekete et al. (2007) for higher-dimensional bin-packing problems (e.g., 2DBPP and 3DBPP). The main difference is in the way the feasibility of a bin is tested. They use an enumerative solution approach based on isomorphic packing classes.

A comparison of the aforementioned methods is provided in Table 1. Note that Junqueira et al. (2012), Paquay et al. (2016), and Fekete et al. (2007) generated and used their own instances, rather than the Martello et al. (2000) benchmark instances. There are two main issues with the proposed approaches. The exact methods are computationally slow. Solving a medium-size problem to optimality is still computationally very challenging. Heuristics, on the other hand, consider only a limited number of options or disregard basic practical constraints such as bin stability and item support. We believe the methodology we propose is a

Table 1. Performance of the Exact Methods

Methodology	Maximum number of items	Average optimality gap	Percent of instances solved to optimality, %
Martello et al. (2007)	50	—	89
Fekete et al. (2007)	80	—	72
Hifi et al. (2010)	90	21.27	—
Junqueira et al. (2012)	100	1.77	50
Paquay et al. (2016)	27	8.37	54

serious attempt at using exact approaches, namely, column generation to solve large problem instances while being able to accommodate practical constraints such as item support.

3. Three-Dimensional Packing Problems with Layering

Layer-building approaches for packing problems were introduced by Bischoff and Ratcliff (1995). Lodi et al. (2004) proposed layer-based mathematical models for the 2DBPP and two-dimensional strip-packing problem (2DSPP) and introduced new combinatorial lower bounds that can be computed in O($n\log n$), where n is the number of items. Based on these models, Bettinelli et al. (2008) developed a branch-and-price for the 2DSPP. Finally, Cui et al. (2017) devised a solution approach, where they use column generation and a residual algorithm to obtain one-dimensional layers. These papers generally build layers with single type items, or do not generate layers with homogeneous height, undercutting the real-world applicability of the solutions. Additionally, they work on simpler two-dimensional problems and do not allow for multiple items to be stacked vertically inside each layer. In our approach, however, we consider complex two-dimensional layers individually in a column-generation procedure, allowing vertical stacking within *layers* using *superitems*. Being important in our approach, we start by defining these two notions.

A layer is defined as a two-dimensional arrangement of items within the horizontal boundaries of a bin with no items stacked on top of each other. The way layers are defined does not allow for multiple stacked items. Consequently, it ignores certain arrangements of items. This issue is clearly visualized in Figure 2, where the more compact arrangement in (b) is ignored within the layer definition, whereas the arrangement in (a) is composed of two layers. To overcome this, we use the concept of superitems. A superitem is a collection of individual items that are compactly stacked together. A superitem is useful, as it is a stable arrangement of items by design, and it provides support to items placed on top as single items do.

To model the 3DBPP using a set-covering formulation, let items be indexed by $i \in \mathscr{I}$, layers by $l \in \mathscr{L}$, and bins by $b \in \mathscr{B}$. Superitems are treated as new items and are added to set $\mathscr{I}$ to form set $\mathscr{CI}$, which includes both the original items $i \in \mathscr{I}$ and the superitems $s \in \mathscr{SI}$. Let us define columns with binary parameters $\bar{z}_{sl}$ and f_{si} that specify whether superitem s is included in layer l and whether superitem s includes item i, respectively, and continuous parameter $\bar{o}^l$ that gives the height of layer l, as well as binary variables t_b that take value 1 if bin b is used, u_{lb} that take value 1 if layer l is assigned to bin b, and α_l that take value 1 if layer l is selected. The layer-based 3DBPP formulation is

$$[BPPLS]: \min \sum_{b \in \mathscr{B}} t_b$$

$$\text{s.t.} \sum_{l \in \mathscr{L}} \sum_{s \in \mathscr{CI}} f_{si} \bar{z}_{sl} \alpha_l \geq 1 \qquad\qquad i \in \mathscr{I}, \qquad\qquad (1)$$

$$\sum_{l \in \mathscr{L}} \bar{o}^l u_{lb} \leq H \qquad\qquad b \in \mathscr{B}, \qquad\qquad (2)$$

$$u_{lb} \leq t_b \qquad\qquad b \in \mathscr{B}, l \in \mathscr{L}, \qquad\qquad (3)$$

$$u_{lb} \leq \alpha_l \qquad\qquad b \in \mathscr{B}, l \in \mathscr{L}, \qquad\qquad (4)$$

$$\alpha_l, t_b, u_{lb} \in \{0, 1\} \qquad\qquad l \in \mathscr{L}, b \in \mathscr{B}.$$

Constraints (1) ensure that each item is covered. Constraints (2) ensure that the total height of the layers assigned to a bin does not exceed the height of the bin. Constraints (3) and (4) make sure that a layer is assigned to a bin only if it is selected and the bin is used.

Figure 2. Better Utilization of the Height Dimension Using Superitems

Item placement without superitems   Item placement with superitems
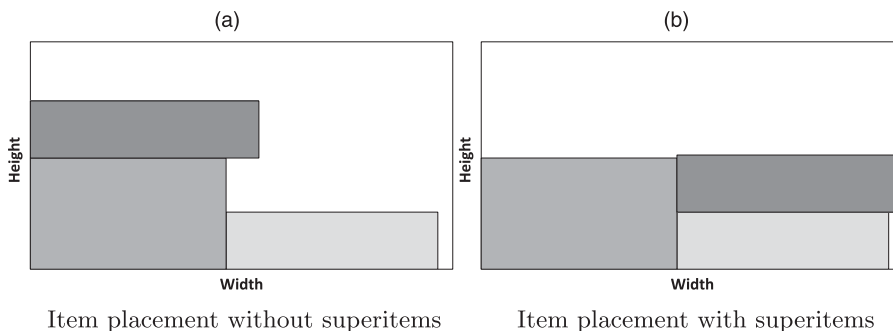
4. Solution by Column Generation

A layer is a two-dimensional arrangement of items and does not depend on the specific bin that it is placed in. Hence, we use the 3DSPP to generate them, as, unlike the 3DBPP, it does not include the extra complication of assigning layers to bins. We hope to generate promising layers efficiently using 3DSPP and then use them to construct solutions to the 3DBPP. Unfortunately, an optimal solution to 3DBPP is not guaranteed. Computational testing shows that the overall approach finds high-quality 3DBPP solutions in very short times (Table 7).

To detail the full solution framework, we start with an MIP formulation for 3DSPP. For that, let items i have width w_i, depth d_i, and height h_i, superitems s have width w_s and depth d_s, and consider an unlimited number of identical containers (bins) $b \in \mathcal{B}$ with width W, depth D, and height H. Let continuous variables c_i^1 and c_i^2 represent the width and depth coordinates of the front, left, bottom corner of item i, and binary variables x_{ij}, y_{ij} take value 1 if item i precedes item j along the width and depth directions, respectively. Let binary variables z_{sl} take value 1 if superitem s is in layer l and continuous variable o^l to represent the height of layer l. The 3DSPP formulation with superitems is

$$[SPPSI] : \min \sum_{l \in \mathcal{L}} o^l$$

$$\text{s.t.} \sum_{l \in \mathcal{L}} \sum_{s \in \mathcal{CI}} f_{si} z_{sl} = 1 \qquad\qquad i \in \mathcal{I}, \tag{5}$$

$$o^l \geq h_s z_{sl} \qquad\qquad s \in \mathcal{CI}, l \in \mathcal{L}, \tag{6}$$

$$\sum_{s \in \mathcal{CI}} w_s d_s z_{sl} \leq WD \qquad\qquad l \in \mathcal{L}, \tag{7}$$

$$x_{sj} + x_{js} + y_{sj} + y_{js} \geq z_{sl} + z_{jl} - 1 \qquad j > s : s, j \in \mathcal{CI}, l \in \mathcal{L}, \tag{8}$$

$$x_{sj} + x_{js} \leq 1 \qquad\qquad j > s : s, j \in \mathcal{CI}, \tag{9}$$

$$y_{sj} + y_{js} \leq 1 \qquad\qquad j > s : s, j \in \mathcal{CI}, \tag{10}$$

$$c_s^1 + w_s \leq c_j^1 + W(1 - x_{sj}) \qquad s \neq j : s, j \in \mathcal{CI}, \tag{11}$$

$$c_s^2 + d_s \leq c_j^2 + D(1 - y_{sj}) \qquad s \neq j : s, j \in \mathcal{CI}, \tag{12}$$

$$0 \leq c_s^1 \leq W - w_s \qquad\qquad s \in \mathcal{CI}, \tag{13}$$

$$0 \leq c_s^2 \leq D - d_s \qquad\qquad s \in \mathcal{CI}, \tag{14}$$

$$x_{sj}, y_{sj} \in \{0, 1\} \qquad\qquad s \neq j : s, j \in \mathcal{CI}, \tag{15}$$

$$z_{sl} \in \{0, 1\} \qquad\qquad l \in \mathcal{L}, s \in \mathcal{CI}, \tag{16}$$

$$o^l \geq 0 \qquad\qquad l \in \mathcal{L}. \tag{17}$$

Constraints (5) ensure that every item is included in a layer. Constraints (6) define the height of layer l. Constraints (7) are redundant valid cuts that force the area of a layer to fit within the area of a bin. They proved useful in speeding up the solution. Constraints (8) enforce at least one relative positioning relationship between each pair of items in a layer. Constraints (9) and (10) ensure that there is at most one spatial relationship between items i and j along each of the width and depth dimensions. Constraints (11) and (12) are the nonoverlapping constraints. Constraints (13) and (14) ensure that items are placed within the boundaries of the bin. Unless all possible superitem combinations are considered, the formulation does not guarantee optimal solutions to 3DSPP. However, our computational experiments show that even with a limited number of superitems, the proposed approach performs really well. Details on how to generate superitems for both theoretical and practical implementations are given in Section 5.

Being a large MIP with many binary variables, the direct solution of [SPPSI] using a commercial solver is found to take up to 5 minutes for instances with up to 20 items. Without valid inequalities (7), the solution takes 30 minutes. However, even the smallest of the industry instances ($|\mathcal{I}| = 50$) is not solvable within 24 hours. To remedy this, we adopt a column-generation framework. In fact, [SPPSI] is developed with this in mind. Using the same parameters and variables as in [BPPLS], the set covering formulation for 3DSPP is

$$[SPPLS] : \min \sum_{l \in \mathcal{L}} \bar{o}^l \alpha_l$$

$$\text{s.t.} \sum_{l \in \mathcal{L}} \sum_{s \in \mathcal{CI}} f_{si} \bar{z}_{sl} \alpha_l \geq 1 \qquad i \in \mathcal{I},$$

$$\alpha_l \in \{0, 1\} \qquad l \in \mathcal{L}.$$

The linear relaxation of [SPPLS] is the Dantzig–Wolfe master problem. When defined on a subset of layers $\mathcal{L}'$, we obtain the restricted master problem:

$$[RMP] : \min \sum_{l \in \mathcal{L}'} \alpha_l \overline{o}^l$$

$$\text{s.t.} \sum_{l \in \mathcal{L}'} \sum_{s \in \mathcal{C}\mathcal{I}} f_{si} \overline{z}_{sl} \alpha_l \geq 1 \qquad i \in \mathcal{I}, \tag{18}$$

$$\alpha_l \geq 0 \qquad\qquad l \in \mathcal{L}'.$$

Let λ_i, $i \in \mathcal{I}$ be the dual variables corresponding to constraints (18). The reduced cost of a new layer l is $o^l - \sum_{i \in \mathcal{I}} \sum_{s \in \mathcal{C}\mathcal{I}} \lambda_i f_{si} z_{sl}$. The pricing subproblem to generate new layers (columns) is

$$[SP] : \min \sum_{l \in \mathcal{L}} \left(o^l - \sum_{i \in \mathcal{I}} \sum_{s \in \mathcal{C}\mathcal{I}} \lambda_i f_{si} z_{sl} \right) \tag{19}$$

$$\text{s.t.} \ (6) - (17),$$

which separates by l. The solution $(\overline{o}^l, \overline{z}_{sl})$ forms a column that is added to [RMP]. If the subproblem fails to generate columns, [RMP] provides a lower bound to [SPPLS].

4.1. Solution of the Pricing Subproblem

The subproblem [SP] takes less than 1 minute to solve using CPLEX, when the number of items considered is 20 or less. As the number of items increases, the solution time increases significantly. For example, when $|\mathcal{I}| > 100$, CPLEX takes more than 1 hour of CPU time. Because [SP] is solved repeatedly, such times are not acceptable. To overcome this, we suggest to solve a relaxed version where placement is ignored. We exploit the property that the objective function of [SP] is only a function of the items in the layer (variables z_{sl}) and not on the placement (variables c_s^1 and c_s^2) to suggest an iterative algorithm to solve [SP].

The algorithm first ignores the placement constraints (8)–(17) and finds a set of items such that the reduced cost is minimized. If it is nonnegative, the column-generation algorithm terminates. Else, a set of items $\mathcal{S}$ is found such that the reduced cost is negative. The algorithm then checks whether there is a feasible placement of $\mathcal{S}$ in a layer, by solving [SP] for the set $\mathcal{S}$ instead of $\mathcal{I}$ until a feasible solution is found. If such a feasible placement exists, the algorithm terminates, and the column is added to the master problem. Else, a feasibility constraint $\sum_{s \in S} z_{sl} \leq |S| - 1$ is added to the relaxed subproblem, and the iterative algorithm continues. Even when $|\mathcal{I}|$ is large, $|\mathcal{S}|$ is expected to be much smaller than $|\mathcal{I}|$, which makes the feasibility check fast. Based on our experiments, even for problems with $|\mathcal{I}| > 100$, the iterative algorithm succeeds in finding good layers within an average of 1 minute.

To improve the solution time, [SP] could also be solved heuristically. There are numerous well-performing 2D packing algorithms in the literature. In this work, we use the Maxrects algorithm (Jylänki 2010) because it is readily available as a C++ library and was found to perform well in our experiments. Maxrects sorts items in decreasing order of their area and places them one at a time in a fashion similar to the extreme point heuristic (Crainic et al. 2008). We take advantage of the sequential item placement that Maxrects adopts to make use of the dual variables (λ_i). Because the reduced cost of a layer l is $o^l - \sum_{i \in \mathcal{I}} \sum_{s \in \mathcal{C}\mathcal{I}} \lambda_i f_{si} z_{sl}$, items with higher dual variables are more likely to minimize the reduced cost. Therefore, we sort the items in decreasing order of λ_i and use Maxrects to place them. In generating the layers, we take advantage of the multiple placement rules that Maxrects offers (e.g., short-side fit, long-side fit, area fit, bottom-left rule, and corner point). Experimentally, a layer is generated in less than 0.0001 seconds on average for instances with up to 3,000 items. The resulting feasible solutions with negative reduced costs, whether found through the relaxed or heuristic approaches, are added to the pool of columns in [RMP].

4.2. Column-Generation and Branch-and-Price Frameworks for 3DSPP

The column-generation framework for 3DSPP starts with a set of columns to warm-start the procedure and then iterates between the subproblem [SP] and reduced master problem [RMP] until the lower bound corresponding to the linear-programming relaxation of [SPPLS] is found. At the warm start, columns are generated heuristically using Maxrects with random dual variables. This concludes node 0, where a set of columns, their corresponding α_l values, and a lower bound are available. Feasible solutions are identified when α_l are binary. For large instances of 3DSPP, and given that the subproblem is not solved to optimality, column generation is terminated when no columns with negative reduced cost can be generated or when the objective value of [RMP] is not improved for 20 consecutive iterations. To solve [SPPLS] to prove optimality, we incorporate the

column-generation procedure in a branch-and-price scheme. We use the Ryan and Foster (1981) branching rule, where two items are either forced to belong to the same layer or to different layers. To find the constraints to branch on, we scan the coefficient matrix of $[RMP]$ to identify two items m and n and two layers (columns) where m and n are both in the first layer, but only one is in the second layer. The columns with the highest α_l values are picked first. In the left child node, m and n are forced to be together, whereas in the right child node, at most one is allowed (see Vance et al. 1994 for more details). Because the left child leads to an easier problem to solve, we use a depth-first search through the left child when traversing the branch-and-bound tree. The branching constraints can easily be applied to the column generation subproblem. To generate columns in the left child node, we add a constraint in the form of $\sum_{s \in \mathcal{CI}} f_{sm} z_{sl} = \sum_{s \in \mathcal{CI}} f_{sn} z_{sl}$. For the right child node, the constraint to be added is $\sum_{s \in \mathcal{CI}} (f_{sm} + f_{sn}) z_{sl} \leq 1$, where we remove the superitems that include both items m and n from $\mathcal{CI}$.

4.3. Bin Construction Heuristic for 3DBPP

Once a solution to 3DSPP is found, whether using branch-and-price or column generation at the root node, selected layers are used to construct a solution to the 3DBPP. For this, the layers are first sorted in decreasing density and are stacked sequentially. If a layer cannot fit in a bin, a new bin is opened. A flowchart for the full approach is depicted in Figure 3, where on the left side, the branch-and-price and column-generation procedures for 3DSPP are detailed, whereas on the right side, the layer-based column-generation heuristic for the 3DBPP is described. The bin-construction heuristic selects layers ordered in descending order of density, $(\sum_{i \in \mathcal{I}} w_i d_i / WD)$, until all items are covered (Algorithm B.1). Layers with higher densities are prioritized, as they implicitly improve bin stability. Based on our tests, many of the layers with high density have common items. To be able to select more of these layers, we allow each item to be covered at most three times. We also let each selected layer to have a maximum of three items that are covered by using the previously selected layers. To ensure that each item is covered only once, we keep the items that are covered multiple times only in the layer with the highest α_l value and remove them from the rest of the selected layers. We then place the remaining items in an empty layer using the Maxrects heuristic. Because the removal of items creates empty space, we also try to place items that are not covered yet in this layer to further improve item coverage and increase layer density (Algorithm C.1). The bin-construction procedure is given in Algorithm D.1. Unfortunately, not all items are covered by layers. There are two main reasons for this. First, layers with less than 50% density are discarded, and, second, some items may have heights that are too different from the rest and cannot be grouped in a layer in a stable manner. Therefore, we build the majority of bins using layers and place the remaining items on top in an S-shaped placement (see Figure 4) by alternating between short and tall items (Algorithm E.1). If the majority of the bin is filled with layers and the remaining items occupy only a small

Figure 3. Overall Solution Approach
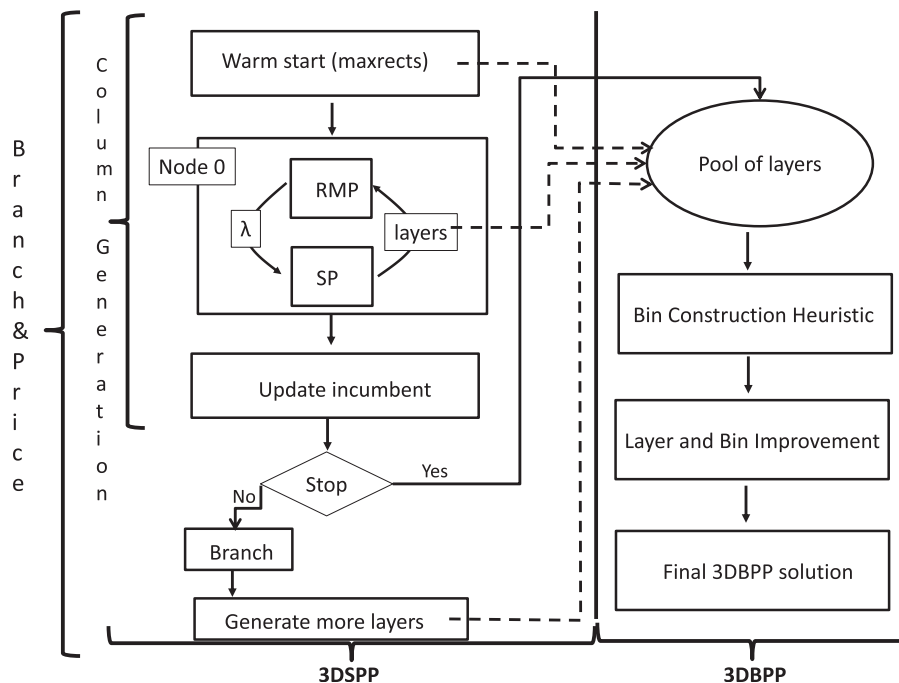
Figure 4. S-Shaped Placement
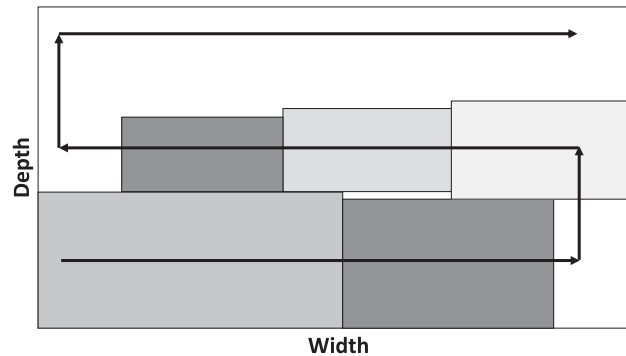
portion at the top, the constructed bin would be stable. Minor instabilities can be fixed through shrink wrapping (i.e., horizontally compressing the pallet by wrapping stretch film around it), which is standard in the industry. In our testing, 70% of the bin is built by using layers.

5. Layer- and Bin-Improvement Strategies

To construct dense and stable layers that are crucial in finding practical solutions to 3DBPP, we propose three enhancement strategies. The first strategy is based on the formation of superitems. The second is a post-processing strategy through item replacement. The third is a layer-reorganization and spacing strategy.

5.1. The Formation of Superitems

For stability, industry rules require a minimum percentage of the surface area of an item to be supported by the item(s) underneath it. As suggested by our industry partner, we used 70%. To satisfy this, superitems are formed by stacking a large item on top of an almost equal item [see Figure 5(a)]. Additionally, superitems are formed by placing identical items horizontally as in Figure 5, (b) and (c). Figure 5(d) shows a mixed formation. In our implementation, we use two- and four-item horizontal and two-item vertical formations. Although increasing the number of items per formation provides many more potential placements, the computational effort increases significantly. Algorithm F.1 describes the superitem generation in more detail.

5.2. Item Replacement

The item-replacement strategy is a simple procedure that allows the generation of additional layers with minimal computational effort. The replacement is limited to items that satisfy the minimum industry overlap in surface area with the item being replaced. The approach is described in Algorithm G.1.

5.3. Layer Reorganization and Spacing

The layer-spacing strategy aims to space items evenly to provide better support to layers. One way to space items is to fix the relative positioning variables in constraints (8)–(17) and solve a mathematical model with the objective of maximizing the minimum distance between the items in each dimension. This, however, is not effective, as empty spaces are not distributed evenly throughout a layer. Alternatively, we propose an approach based on linear programming. When an item moves in one direction (width or depth), it may impact other items in that direction. Let a segment S_i be the set of items that may be impacted by moving item i in

Figure 5. Superitem Formations

(a) (b) (c) (d)

Vertical 2-item horizontal 4-item horizontal Mixed
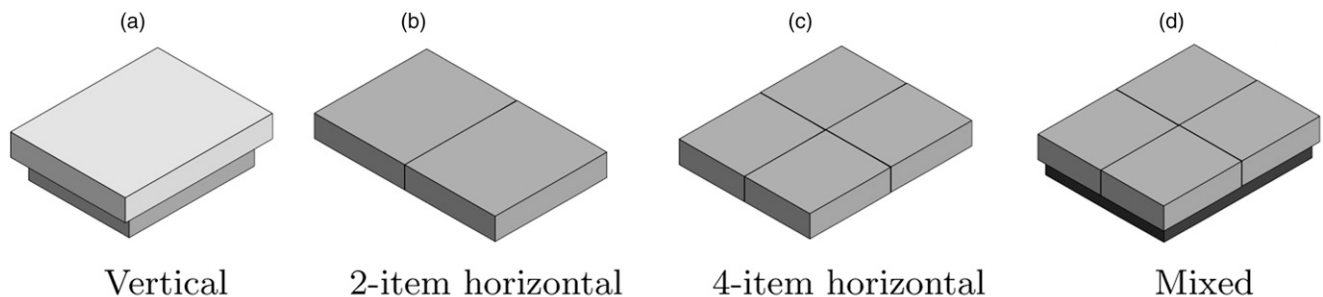
some direction. For the width dimension, the complement to the set $\{j \in \mathcal{S} : (c_j^2 > c_i^2 + d_i) \cup (c_j^2 + d_j < c_i^2)\}$ defines segment S_i. Figure 6 illustrates this concept, where the spacing of item 5 in the width dimension can only be affected by items 1, 4, 6, and 7. Note that we only consider unique segments, and we only use the distances between adjacent items. Additionally, a segment is discarded if it is a subset of another segment.

Because segment definitions in one dimension may change with spacing in the other dimension, we apply spacing to each dimension sequentially. The segmented spacing formulation in the width dimension is

$$
\begin{aligned}
\max \quad & \sum_{s \in \mathcal{S}} a_s^1 \\
\text{s.t.} \quad & c_i^1 + w_i \le c_j^1 + W(1 - \bar{x}_{ij}) && i \ne j, i, j \in \mathcal{S}, \\
& 0 \le c_i^1 \le W - w_i && i \in \mathcal{S}, \\
& a_s^1 \le c_j^1 - (c_i^1 + w_i) && i \ne j, \bar{x}_{ij} = 1, i, j \in \mathcal{S}, s \in S, \\
& c_i^1 \ge 0 && i \in \mathcal{S},
\end{aligned}
$$

where S is the set of segments along the width dimension, and a_s^1 are continuous decision variables denoting the minimum distance between the items in segment s. A similar model is used to space in the depth dimension.

6. Practical Requirements

In practice, three-dimensional bin packing has to satisfy important practical constraints. In mixed-case palletization and container loading, item support, bin stability, and load bearing are particularly important. The proposed layering approach is suitable to deal with some of these constraints, either explicitly or implicitly, when the layers are formed and selected to create feasible solutions.

To ensure item support, we divide the set of items $\mathcal{I}$ into item groups $\mathcal{I}_g$ that only contain items of the same height and generate layers based on them. We allow a tolerance of 5 mm, which is an industry standard. It is important to note that an item may be in multiple item groups $\mathcal{I}_g$ based on its height. Another important practical constraint is family grouping. Items in the same family group (e.g., beverages) should preferably be packed in close proximity and in the same bin, which may also be handled by using $\mathcal{I}_g$. Finally, load bearing, an important practical constraint, may be accommodated by limiting the subsets $\mathcal{I}_g$ to only contain items with similar load-bearing capabilities. To ensure bin stability, layers are placed in the bins in decreasing order of load-bearing capability. This approach decomposes the layer-generation subproblem into multiple smaller subproblems, one for each group, which results in a considerable decrease in solution time. Finally, we restrict the number of sets $\mathcal{I}_g$ by eliminating groups with $\sum_{i \in \mathcal{I}_g} w_i d_i \le 0.5WD$, because they do not lead to dense layers.

7. Computational Experiments

We perform extensive computational testing of the proposed solution approach and compare it to the best solution methodologies. We start by describing standard benchmark instances from the literature and then provide a new and realistic benchmark data set. We compare our approach with the best-performing heuristic and exact approaches from the literature on the standard instances and provide a comparison with algorithm 864 of Martello et al. (2007) on the new instances. Testing is conducted on a computer with Intel i7-5500U CPU,

Figure 6. Visual Representation of the Segments
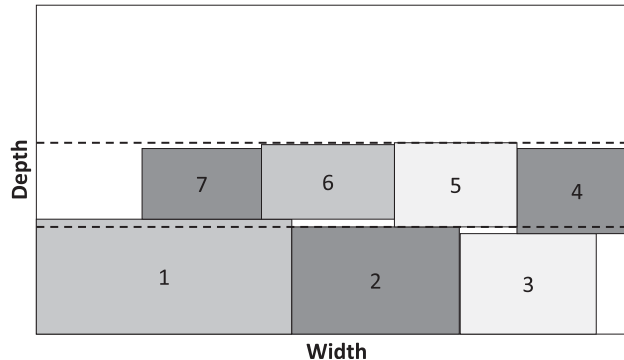
with 2.40 GHz and 16 GB of RAM on Windows 8.1. The mathematical models are solved by using CPLEX Concert technology with CPLEX version 12.6.1. The algorithms are coded in C++ by using Visual Studio 2012 IDE.

7.1. Test Data

There is only one standard 3DBPP data set based on the random instance generator of Martello et al. (2000). The set has 9 classes, with 50, 100, 150, and 200 items. We generate 360 instances, 10 for each class and item combination. The random instance generator uses the same seed, so all generated instances are guaranteed to be the same as those used in the literature.

We also obtained 342 real-life data sets from an industrial partner, with number of items ranging from 50 to 3,000. All of these sets use bin dimensions of $W = 1,200$ mm, $D = 800$ mm, $H = 2,055$ mm. These data sets contain "mixed cases," meaning that there is a high item variety and few identical items.

When comparing the number of items per bin and the ratio of item to bin dimensions in the industry data set and in the random instances, it is obvious that the latter are far from being realistic (Figure 7). The main issue is that item sizes are large relative to bin size. Looking at the lower bounds, the average number of items per bin is 4.31 in Martello et al. (2000) instances, compared with at least 90 items per bin for the real-life data sets. The same was found by Zhu et al. (2012b), leading to them using a secondary real-life data set to test their algorithm. To best of our knowledge, there is no 3DBPP instance generator that produces realistic instances. To remedy this, we propose a new realistic instance generator.

Some of the important characteristics of items are the ratio of item depth and height to its width, volume, and frequency of occurrence. The first two characterize the dimensions of an item. The reason for considering the proportions and the volume of an item instead of its dimensions is to construct generalized items with different sizes whose dimensions are properly related to each other.

The item characteristics are determined based on the industry data to which we had access. A total of 166,406 items with 73,978 distinct items are used to construct distribution functions that form the basis of the proposed generator. Figure 8 shows histograms of the item characteristics. A distribution fitting package in *R*, called "fitdistrplus" (Delignette-Muller and Dutang 2015), is used. It provides the fitting and comparison of different probability distribution functions with goodness-of-fit tests, such as Kolmogorov–Smirnov, Cramer

Figure 7. (Color online) Comparison Between a Benchmark and an Industry Instance

A Martello et al. (2007) instance

An industry instance
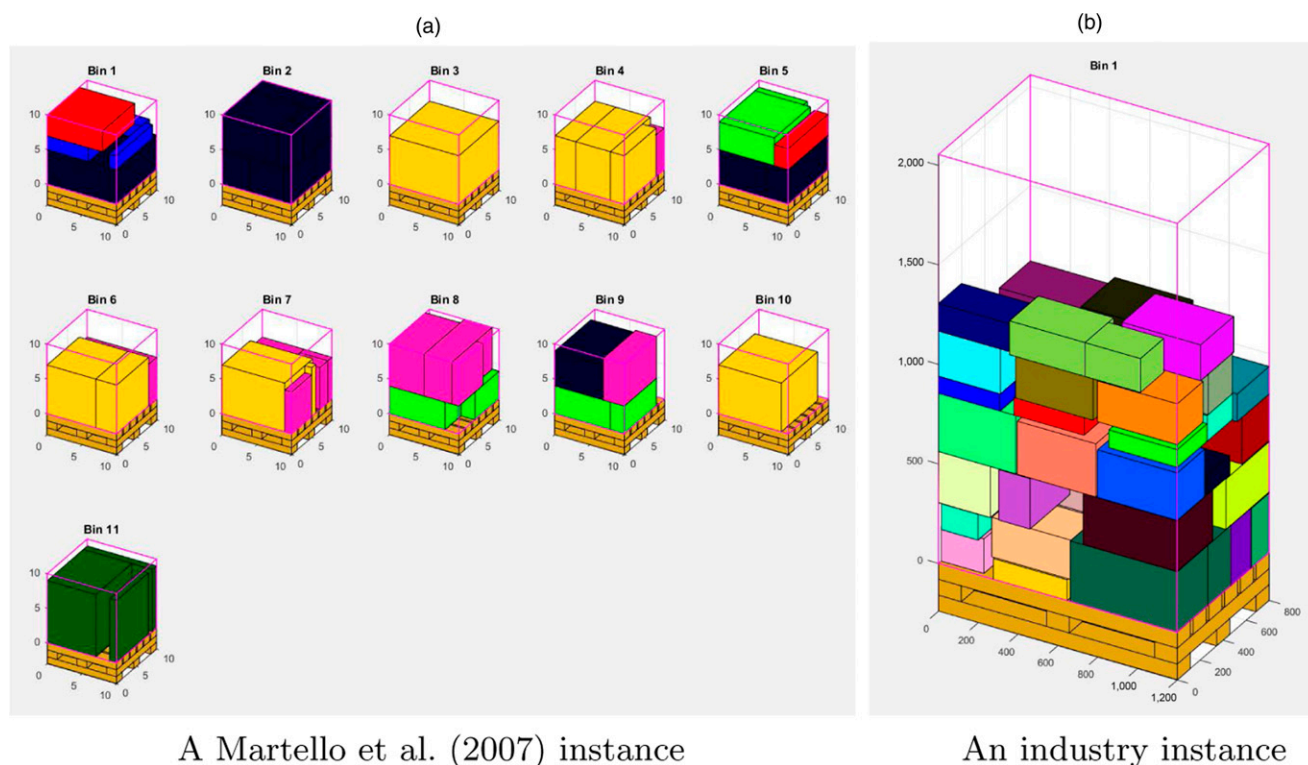
Figure 8. (Color online) Item Characteristics

Histogram for d/w ratio

Histogram for h/w ratio

Histogram for item volumes

Histogram for item repetitions
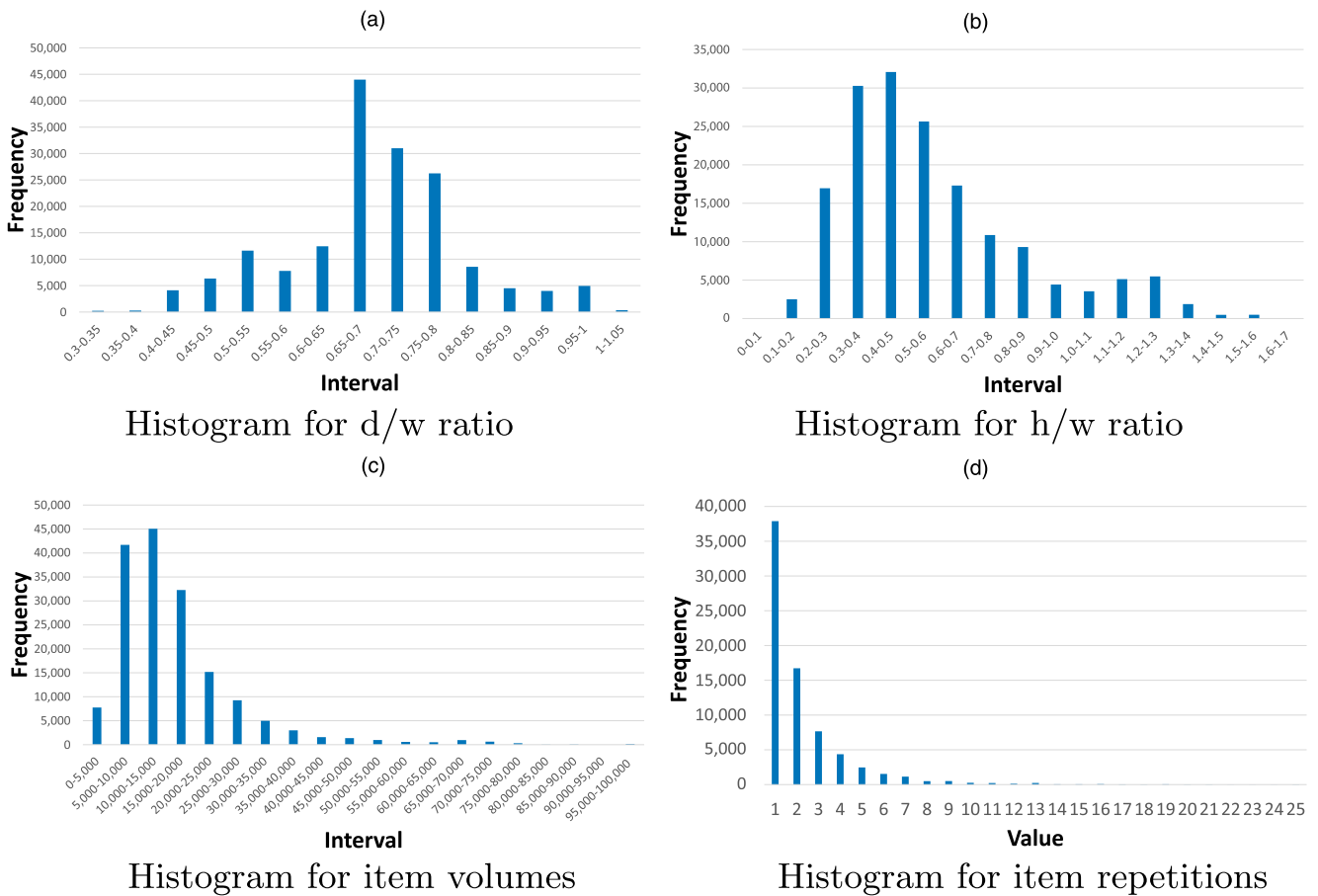
von Mises, and log-likelihood. Based on these tests, the best probability distribution functions and their parameters are given in Table 2.

For the volume characteristic, we apply *k*-means clustering to assess its distribution. Varying *k*, we find that the sum of squared errors is the lowest for *k* = 5. The analysis separates the 166,406 items into five categories. Category 1 has 72,037 items and $v \in [2.72, 12.04]$, category 2 has 55,436 items and $v \in [12.05, 20.23]$, category 3 has 26,254 items and $v \in [20.28, 32.42]$, category 4 has 9,304 items and $v \in [32.44, 54.08]$, and category 5 has 3,376 items and $v \in [54.31, 100.21]$ (in dm^3). Furthermore, we calculated the percentage of items of each category for each industry instance and applied *k*-clustering. The four classes in Table 3 were identified.

The instance generator uses the determined parameters to generate instances with different sizes. The generator is available for use by the academic community, by contacting the corresponding author. We use the generator to create 5 instances of size 50; 100; 150; 200; 500; 1,000; 1,500; and 2,000 for each class, leading to 160 instances in total.

Finally, we applied centroid and average error (distance from the centroid) analyses to validate the similarity of the generated instances to the industry ones. Table 4 reports the results of this analysis, where the columns show the centroid of each class and the average distance of each item from these centroids for both the industry and the generated instances. We used the *scale* function in *R* before calculating the distance

Table 2. Distribution Functions and Parameters for Item Characteristics

Characteristic	Distribution	Parameters
Depth/width ratio	Normal	(0.695, 0.118)
Height/width ratio	Lognormal	(−0.654, 0.453)
Repetition	Lognormal	(0.544, 0.658)

Table 3. Percentage of Each Category of Item in Each Instance Class

	Percentage of items				
Class	Category 1	Category 2	Category 3	Category 4	Category 5
1	28.48	58.75	12.67	0.1	0
2	33.08	32.36	23.34	7.94	3.28
3	66.88	24.75	5.7	2.6	0.08
4	78.58	13.16	6.33	1.78	0.15

Table 4. Industry vs. Generated Data

	Industry		Generated	
Class	Centroid (w, d, h, v)	Average distance	Centroid (w, d, h, v)	Average distance
1	(291.9, 200.34, 269.15, 15.5)	1.76	(338.8, 232.81, 181.95, 13.72)	1.8
2	(362.89, 245.31, 207.32, 19.54)	1.73	(336.89, 231.18, 180.51, 13.53)	1.87
3	(340.77, 236.87, 150.47, 12.53)	1.74	(312.77, 215.31, 168, 10.97)	1.8
4	(320.11, 225.4, 151.6, 12.16)	1.72	(304.86, 209.66, 164.87, 10.19)	1.74

values, so that the dimensions have the same weight. We also scaled and calculated the distance between the centroids of the industry and generated data for each class. The results are 0.39, 0.35, 0.2, and 0.2. Because the differences are closer to 0 than to 1, we can argue that the means of both sets are similar. We then examine the difference between average distances for each class, which are 0.04, 0.14, 0.06, and 0.02, respectively. Closer average distances between data sets demonstrate the similarity of the variance of observations. Because both the means and the variances of the sets are fairly comparable to one another, we conclude that the two data sets are not significantly different from one another.

7.2. Branch-and-Price Results

The branch-and-price approach is tested on 50-item instances from Martello et al. (2000) and the generated instances. Table 5 displays the average number of bins used, the average number of nodes searched, and the average CPU time in seconds. For comparison, the layer-based column-generation heuristic (LCGA) is included. The results show that branch-and-price method finds optimal solutions to 3DSPP within a maximum of 822.6 seconds. Additionally, it finds the optimal fairly close to the root node, which shows us that the gap at the root node is small. On the other hand, LCGA finds good solutions in less than 1 second. As branch-and-price is not expected to handle large instances, we focus on LCGA for the rest of the experiments.

Table 5. Branch-and-Price Results

	Branch-and-price			LCGA		
Class	Average bin	Average nodes	Average CPU (s)	Average bin	Gap (%)	Average CPU (s)
Martello 1	13	5.7	378.4	13.2	1.5	0.56
Martello 2	13.1	4.2	349.2	13.3	1.5	0.68
Martello 3	12.5	4.8	416.7	12.8	2.4	0.41
Martello 4	29.2	3.9	363	29.4	0.7	0.27
Martello 5	7.9	3.4	395.9	8.4	6.3	0.43
Martello 6	9.5	4.1	387.6	9.8	3.2	0.79
Martello 7	7.1	5	448.7	7.4	4.2	0.82
Martello 8	8.9	3.3	465.3	9.2	3.4	0.77
Martello 9	3.8	3.1	418.5	4	5.3	0.91
Generated 1	1	2.2	822.6	1	0	0.24
Generated 2	1	1.7	678.4	1	0	0.19
Generated 3	1	1.9	692.1	1	0	0.17
Generated 4	1	2.1	728.9	1	0	0.13

7.3. Comparison of LCGA to the State-of-the-Art

To assess the efficiency of the proposed layer-based column-generation approach, we compare with the best-performing algorithms in the literature. Zhao et al. (2016) give a comparison of 10 algorithms (Table 6) based on the total number of bins used for classes 1 and 4–8 of Martello et al. (2000). Some classes are omitted, as some approaches do not test on them. The comparison is given in Table 7. The column IbB-2004 gives the best known lower bound due to Boschetti (2004). LCGA outperforms the best algorithm in the literature in 13 out of 24 class-item size combinations and in all classes except 6 and 7. This is likely because classes 6 and 7 have relatively small bin size and large item dimensions, which is far from being realistic and does not favor layering because only few items can fit in one bin. Even though the improvements on the average number of bins used may not seem too large, it is important to note that the overall optimality gap is 2.8%. The fastest methodology in Table 7 is space defragmentation by Zhu et al. (2012b) with a time limit of 30 seconds per instance. In comparison, LCGA uses only 3.17 seconds on average.

In Table 8, we compare with algorithm 864 (MPV-2000) on 100-item instances with standard and larger bin sizes. The larger bin sizes have approximately 10 times the total volume compared with the standard sizes. Increasing bin dimensions reduces the item to bin volume ratio and leads to more realistic data sets. All of the solutions reported for algorithm 864 are obtained by using the provided C code (Martello et al. 1998) with a CPU time limit of 1,000 seconds. In Table 8, LB gives the average lower bound (L_2) over 10 instances, and No. Better provides the number of instances in which LCGA outperforms algorithm 864. The columns Average Bin and CPU report the average number of bins used and the average CPU time for both approaches, respectively.

LCGA outperforms algorithm 864 on all instances based on the average number of bins used. LCGA uses 44.14% less bins than algorithm 864 for large bins and 5.55% for smaller ones. This shows the efficiency of LCGA in solving realistic instances. Additionally, there were no instances where algorithm 864 used less bins than LCGA. LCGA is nearly 500 and 300 times faster for small and large bin sizes, respectively.

7.4. Comparison on Generated Instances

In this section, we compare the performance of LCGA to algorithm 864 on large generated instances. This experiment aims to demonstrate that LCGA is suited to tackle realistic problems with up to 2,000 items. The code provided for algorithm 864 is able to solve instances with up to 100 items, due to a parameter they use for array sizes. We increased this parameter to accommodate larger problems. The results are given in Table 9, where the class, number of items, average lower bound (L_2), average CPU time, average number of bins used, and average support percentage per item are reported for LCGA and algorithm 864, respectively. The "Support (%)" is the percentage of an item's base area that is supported by items underneath it. This metric is important in assessing bin stability and is a requirement in practice. The results are averaged over five instances.

The results show that LCGA clearly outperforms algorithm 864 in terms of number of bins, support, and CPU time. LCGA always finds at most the same number of bins with an overall reduction of about 10%. algorithm 864 reaches the 1-hour time limit in 133 out of the 160 instances, whereas LCGA terminates in all instances. The increase in percentage support is remarkable. LCGA provides 74.14% more vertical support than algorithm 864. The latter is only able to provide the 70% industry minimum in only 27 out of the 160 instances, whereas LCGA achieves this in all 160. Considering that 70% support per item is a current industry requirement, LCGA has a high potential of application for real-life palletization problems.

Table 6. List of Papers and Approaches Used for Comparison

Year	Paper	Code	Name
2000	Martello et al. (2000)	MPV-2000	Algorithm 864
2000	Martello et al. (2000)	MPV-2000-BS	Algorithm 864
2000	Martello et al. (2000)	MPV-2000-Spack	Algorithm 864
2002	Lodi et al. (2002)	L2002-HA	Tabu search
2002	Lodi et al. (2002)	L2002-TS	Tabu search
2003	Faroe et al. (2003)	F-2003	Guided local search
2008	Crainic et al. (2008)	C-2008	Extreme point placement algorithm
2009	Crainic et al. (2009)	C-2009	TS^2PACK
2012	Zhu et al. (2012b)	SD	Space defragmentation
2004	Lower bound (Boschetti 2004)		

Table 7. Detailed Comparison Against the Literature on Standard Instances

Class	Bin dimensions	Item no.	MPV-2000	MPV2000-BS	MPV2000-Spack	L2002-HA	L2002-TS	F-2003	C-2008	C-2009	SD	LCGA	IbB-2004
1	100 × 100	50	13.6	13.5	15.3	13.9	13.4	13.4	13.7	13.4	-	**13.2**	12.9
		100	27.3	29.5	27.4	27.6	26.6	26.7	27.2	26.7	-	**26.1**	25.6
		150	38.2	38	40.4	38.1	**36.7**	37	37.7	37	-	**36.7**	35.8
		200	52.3	52.3	55.6	52.7	51.2	51.2	51.9	51.1	-	**50.7**	49.7
Class total			131.4	133.3	138.7	132.3	127.9	128.3	130.5	128.2	127.4	**126.7**	124
4	100 × 100	50	**29.4**	**29.4**	29.8	**29.4**	**29.4**	**29.4**	**29.4**	**29.4**	-	**29.4**	29
		100	59.1	59	60	59	59	59	59	**58.9**	-	59	58.5
		150	87.2	87.3	87.9	86.9	**86.8**	**86.8**	**86.8**	**86.8**	-	**86.8**	86.4
		200	119.5	119.3	120.3	119	118.8	119	118.8	118.8	-	**118.6**	118.3
Class total			295.2	295	298	294.3	294	294.2	294	293.9	294	**293.8**	292.2
5	100 × 100	50	9.2	9.1	10.2	8.5	8.4	**8.3**	8.4	**8.3**	-	8.4	7.6
		100	17.5	17	17.6	15.1	15	15.1	15.1	15.2	-	**14.6**	14
		150	24	23.7	24	21.4	20.4	20.2	21	20.1	-	**19.8**	18.8
		200	31.8	31.7	31.7	28.6	27.6	27.2	28.1	27.4	-	**26.9**	26
Class total			82.5	81.5	83.5	73.6	71.4	70.8	72.6	71	70.3	**69.7**	66.4
6	10 × 10	50	**9.8**	11	11.2	10.5	9.9	**9.8**	10.1	**9.8**	-	**9.8**	9.4
		100	19.4	22.3	24.5	20	19.1	19.1	19.6	19.1	-	**18.9**	18.4
		150	29.6	32.4	35	30.6	29.4	29.4	29.9	**29.2**	-	29.5	28.5
		200	38.2	40.8	42.3	39.1	37.7	37.7	38.5	37.7	-	**37.4**	36.7
Class total			97	106.5	113	100.2	96.1	96	98.1	95.8	**95.5**	95.6	93
7	40 × 40	50	8.2	8.2	9.3	8	7.5	**7.4**	7.5	**7.4**	-	8.1	6.8
		100	15.3	13.9	15.3	13.3	12.5	**12.3**	13.2	**12.3**	-	12.8	11.5
		150	19.7	18.1	20.1	17.2	16.1	**15.8**	17	**15.8**	-	16.5	14.4
		200	28.1	28	28.7	25.2	23.9	**23.5**	25.1	**23.5**	-	24.3	22.7
Class total			71.3	68.2	73.4	63.7	60	59	62.8	59	**58.4**	61.7	55.4
8	100 × 100	50	10.1	9.9	11.3	9.9	9.3	9.2	9.4	9.2	-	**9**	8.7
		100	20.2	20.2	21.7	19.9	18.9	18.9	19.5	**18.8**	-	18.9	18.4
		150	27.3	26.8	28.3	25.7	24.1	23.9	25.2	23.9	-	**23.8**	22.5
		200	34.9	34	35	31.6	30.3	29.9	31.3	30	-	**29.2**	28.2
Class total			92.5	90.9	96.3	87.1	82.6	81.9	85.4	81.9	81.3	**80.9**	77.8
Total			769.9	775.4	802.9	751.2	732	730.2	743.4	729.8	**726.9**	728.4	708.8

Note. Best results indicated in bold.

Table 8. Comparison of Algorithm 864 and LCGA with Different Bin Size

				LCGA		Algorithm 864	
Class	Bin size ($W \times D \times H$)	Lower bound	#Better	Average bin	CPU (s)	Average bin	CPU (s)
1	$100 \times 100 \times 100$	25.2	8	**26.8**	1.9	27.5	1,000
	$215 \times 215 \times 215$	2	10	**3.2**	6.05	4.9	1,000
2	$100 \times 100 \times 100$	24.2	7	**25.9**	1.97	26.4	1,000
	$215 \times 215 \times 215$	2	8	**3.4**	6.47	4.7	1,000
3	$100 \times 100 \times 100$	24.7	7	**27.1**	2.05	29	1,000
	$215 \times 215 \times 215$	2	10	**3.3**	6.25	4.9	1,000
4	$100 \times 100 \times 100$	57.8	2	**59.1**	1.48	59.2	1,000
	$215 \times 215 \times 215$	3.8	10	**4.5**	2.68	6.5	1,000
5	$100 \times 100 \times 100$	13.2	5	**15.3**	2.28	17.3	1,000
	$215 \times 215 \times 215$	1.8	10	**2.2**	1.37	3.5	1,000
6	$10 \times 10 \times 10$	17.5	6	**18.2**	2.22	19.4	1,000
	$22 \times 22 \times 22$	2	3	**3.1**	0.95	3.4	1,000
7	$40 \times 40 \times 40$	11	4	**13.8**	1.93	15.3	1,000
	$86 \times 86 \times 86$	1.2	4	**2.3**	0.83	2.7	1,000
8	$100 \times 100 \times 100$	17.8	5	**18.9**	3.03	19.9	1,000
	$215 \times 215 \times 215$	2	10	**2.6**	3.18	3.9	1,000
9	$100 \times 100 \times 100$	3	10	**4**	2.05	6.7	1,000
	$215 \times 215 \times 215$	1	10	**1**	0.9	2.4	1,000
	Average	11.78	7.17	**13.04**	2.64	14.07	1,000

Note. Best results indicated in bold.

Finally, in Tables A.1–A.4, we provide detailed computational results for all of the generated instances for the use of the scientific community. The tables report the class, number of items, instance number, lower bound, number of bins used, CPU time, average support percentage per item, and percentage of items with at least 70% support for each instance.

7.5. Vertical Support

One of the benefits of the layering approach is its ability to handle item support relatively easily. Once a layer is formed, the areas requiring support as well as those that may provide support are known. We exploit such information to account for support explicitly and provide preliminary results. For that, we modify the bin construction (Algorithm D.1), the item spacing, and the S-shaped placement algorithms.

In Algorithm D.1, items are placed based on their density. We only check whether items are parts of previous layers. To ensure support, we conduct a second check. A layer is placed only if items have at least 70% of their bottom surface covered by the layer underneath. If not, items are spaced to remedy this. For that, a nonlinear mathematical program is used to maximize the overlap between the current layer and the one below it. The objective maximizes the overlap in the width and depth directions. After all possible layers are used to construct bins, we use the extreme point heuristic (Crainic et al. 2008) for the remaining items.

As the S-shaped placement approach used previously cannot guarantee support, we adopt an extreme point approach that uses a merit function to decide on the placement. The merit function evaluates extreme points and favors items that are placed lower in the bin. Even though the approach is only applied to the top 30% of the bin, all extreme points and empty spaces are considered. Table 10 displays preliminary results on two of the generated instances. The columns report the instance details, CPU time, support percentage, number of items with more than 70% support, number of items that have all four corners supported, and number of bins used, respectively, with and without accounting for vertical support. The bins are displayed in Figures 9 and 10.

According to Table 10, all items are supported. In addition, on average, 85.6% of the items have additional four-corner support. Although there is a slight increase in computational time, the number of bins used did not increase. In an industrial setting, minimizing the number of bins is the ultimate measure. Therefore, being able to achieve stability without increasing the number of bins is remarkable. The results are very promising and clearly demonstrate that the layering approach can easily handle a variety of critical practical constraints.

8. Conclusion and Future Work

Motivated by a pressing need from the automated warehousing industry, we studied the three-dimensional bin-packing problem and its practical counterpart, the mixed-case palletization problem. Unlike previous models in the literature, we proposed a new formulation that inherently lends itself to branch-and-price and

Table 9. Comparison of Algorithm 864 and LCGA

			LCGA			Algorithm 864		
Class	Items	Lower bound	CPU (s)	No. of bins	Support (%)	CPU (s)	No. of bins	Support (%)
	50	1	0.24	**1**	86.70	0.10	**1**	34.13
	100	1	1.38	**1**	87.77	3,600	2	41.14
	150	2	15.50	**2**	89.90	3,600	2	36.37
	200	2	37.46	**2**	91.40	3,600	3	41.14
1	500	4	120.31	**5**	91.98	3,600	5	50.78
	1,000	7.2	218.05	**8.8**	92.67	3,600	10	63.52
	1,500	11	439.27	**12.4**	93.26	3,600	13.6	68.49
	2,000	14	888.66	**16**	92.97	3,600	17.2	73.95
	50	1	0.19	**1**	89.50	0.51	**1**	34.61
	100	1	1.60	**1**	86.84	2,880	1.8	48.57
	150	1.2	15.44	**2**	92.00	3,600	2	38.19
	200	2	55.29	**2**	91.61	3,600	2.8	48.82
2	500	4	144.37	**5**	91.76	3,600	5	60.74
	1,000	7	244.66	**9**	92.43	3,600	9.2	72.04
	1,500	11	585.42	**12**	92.97	3,600	13	78.20
	2,000	14	1,035.09	**16**	92.86	3,600	17.4	73.00
	50	1	0.17	**1**	88.89	0.18	**1**	31.78
	100	1	1.50	**1**	87.46	2,880	1.8	45.35
	150	1	21.00	**1.8**	89.35	3,600	2	30.36
	200	2	70.49	**2**	91.32	3,600	2	33.32
3	500	3	344.60	**4**	91.76	3,600	5	46.10
	1,000	6	424.41	**7**	92.12	3,600	8	58.69
	1,500	9	884.64	**10.6**	92.33	3,600	11.6	61.79
	2,000	12	1,208.16	**14**	92.50	3,600	14.8	74.91
	50	1	0.13	**1**	88.80	0.67	**1**	32.72
	100	1	1.77	**1**	87.88	720	1.2	38.93
	150	1	21.67	**1.2**	89.70	2,880	2	35.37
	200	1.8	75.14	**2**	90.33	3,600	2	50.11
4	500	3	252.94	**4**	91.46	3,600	4.2	47.83
	1,000	6	536.38	**7**	91.17	3,600	7.6	66.16
	1,500	8	944.40	**10**	91.71	3,600	10.4	83.65
	2,000	11	1,462.29	**12.8**	92.20	3,600	14	67.82
Average		4.73	314.14	**5.52**	90.8	2,992.55	6.08	52.14

Note. Best results indicated in bold.

column-generation methodologies. The resulting subproblem is a two-dimensional layer-generation problem that is solved optimally as well as heuristically. Generated layers are further enhanced by using a variety of strategies such as item grouping, item replacement, layer reorganization, and layer spacing.

Generating layers and using them to construct bins/pallets has numerous advantages, both at the packing stage in terms of stability and at the unpacking stage for ease of shelving. In addition, layering can accommodate some of the difficult practical constraints, such as item support and bin stability. We accounted for support and provided results where every item is fully supported. Even when support is not enforced explicitly, the layering approach constructed bins where support is satisfied for the majority of items.

Because of the lack of realistic data sets and given our access to industrial data, we trained a realistic instance generator, which we hope will be used for benchmarking. We conducted extensive numerical testing using standard benchmark instances, as well as the realistically generated ones, and compared against all existing approaches. The proposed approach found better solutions compared with the best-performing

Table 10. Vertical Support Results

Class	No. of items	Instance	CPU (s)	Support (%)	No. supported	No. 4-corner	No. of ins
With vertical support							
1	100	4	2.87	94.52	100	84	1
1	1,000	5	299.29	95.17	1,000	872	8
Without vertical support							
1	100	4	1.75	87.57	72	51	1
1	1,000	5	240.99	92.76	826	634	8

approach in the literature on most of the standard benchmark instances, both in terms of number of bins used and bin stability within significantly shorter computational times. It was only outperformed by the space-defragmentation approach when the instance solved did not inherently have layers.

Future work will focus on explicitly accounting for practical constraints required by the industry. In addition to our preliminary work on vertical support, the layering approach can handle other constraints such as planogram sequencing, reduced item support, bin weight limit, and, most importantly, load bearing. These industry requirements may be accounted for during the bin construction process using sophisticated approaches for layer spacing and item placement.

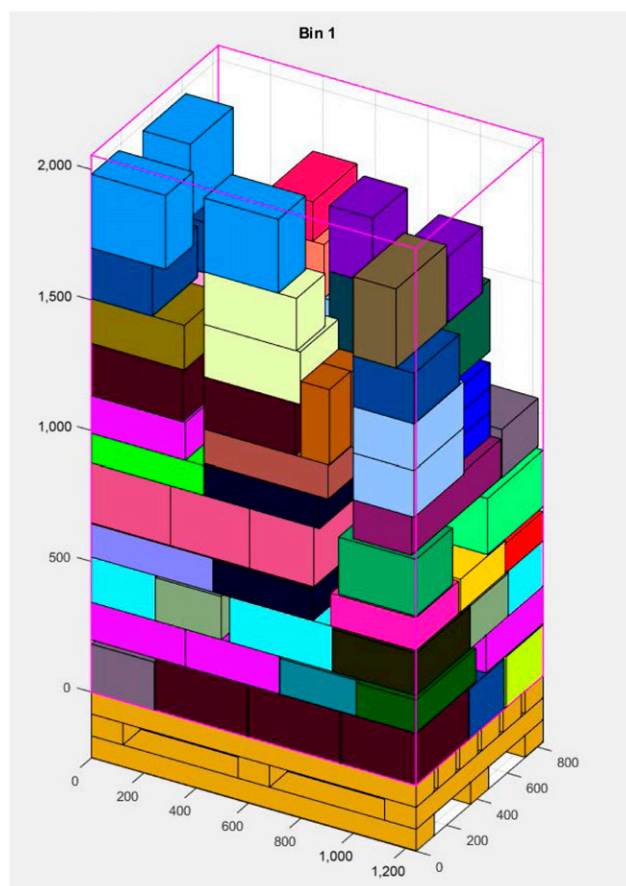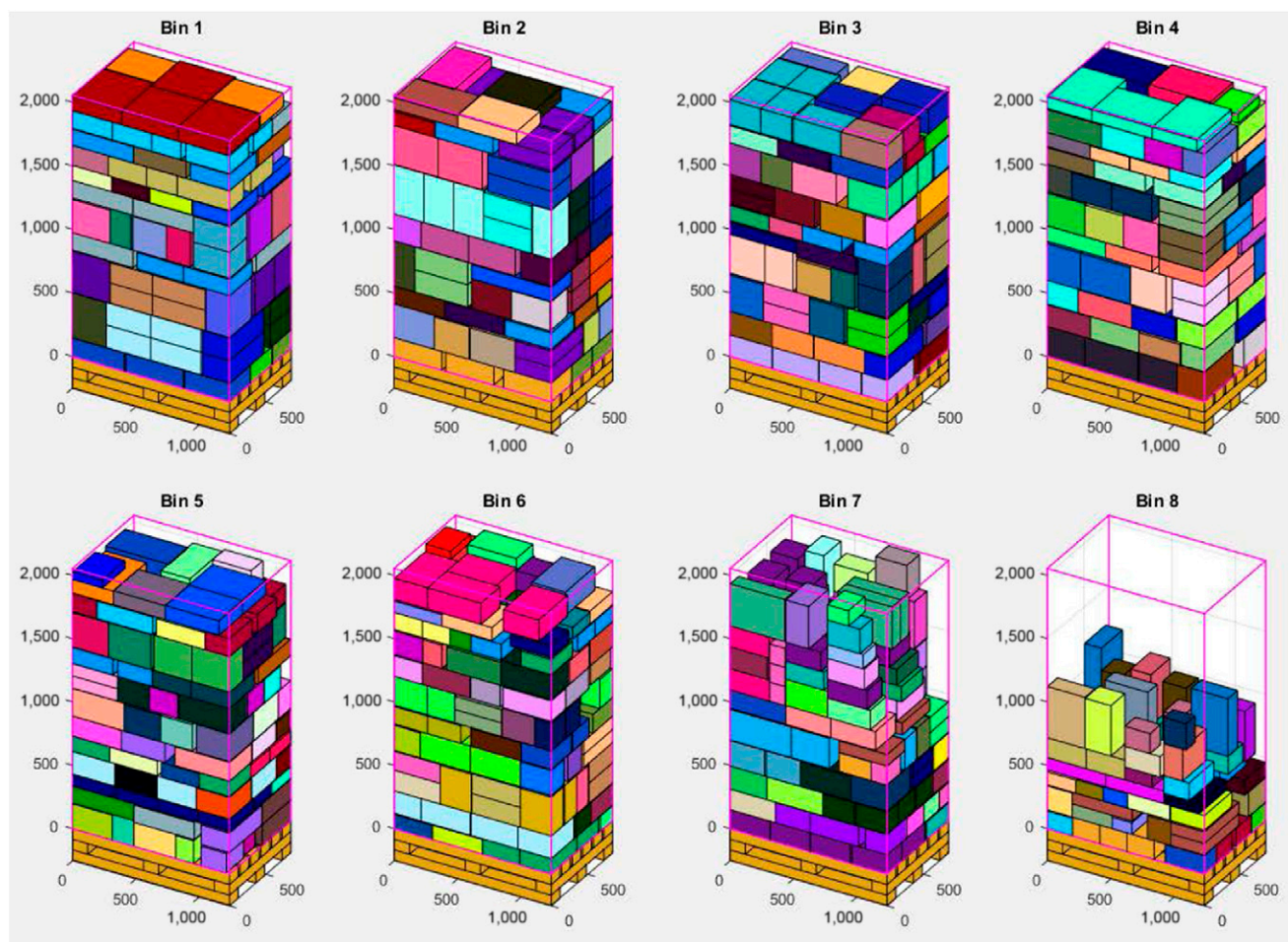**Figure 9.** (Color online) Vertical Support Solution for Class 1, 100 items, Instance 4

Figure 10. (Color online) Vertical Support Solution for Class 1, 1,000 Items, Instance 5

Appendix A. Detailed Results on the Generated Benchmark Instances

Table A.1. Generated Instances: Class 1

No. of items	Instance no.	CPU (s)	Lower bound	No. of bins	Support (%)	Items supported (%)
50	1	0.10	1	1	83.94	58.00
50	2	0.39	1	1	88.68	58.00
50	3	0.26	1	1	87.92	60.00
50	4	0.29	1	1	84.02	70.00
50	5	0.16	1	1	88.93	70.00
100	1	1.03	1	1	88.04	81.00
100	2	1.30	1	1	86.80	77.00
100	3	1.43	1	1	87.04	76.00
100	4	1.75	1	1	87.57	72.00
100	5	1.38	1	1	89.40	78.00
150	1	10.91	2	2	87.67	70.00
150	2	20.30	2	2	91.93	76.00
150	3	12.90	2	2	88.84	75.33
150	4	12.08	2	2	90.70	72.00
150	5	21.31	2	2	90.36	79.33
200	1	55.98	2	2	91.21	81.50
200	2	36.41	2	2	90.41	77.50
200	3	30.26	2	2	90.84	81.50
200	4	41.14	2	2	92.87	88.50
200	5	23.50	2	2	91.67	79.50
500	1	79.15	4	5	91.48	80.40
500	2	148.36	4	5	92.17	79.60
500	3	171.75	4	5	92.21	80.80
500	4	104.23	4	5	91.99	81.00
500	5	98.08	4	5	92.05	83.20
1,000	1	226.20	7	9	92.47	83.70
1,000	2	216.53	7	9	92.63	82.70
1,000	3	215.23	8	9	92.46	83.80
1,000	4	191.28	7	9	93.01	83.60
1,000	5	240.99	7	8	92.76	82.60
1,500	1	480.76	11	13	93.02	84.27
1,500	2	426.49	11	12	93.32	86.47
1,500	3	438.23	11	13	93.27	84.93
1,500	4	433.17	11	12	93.36	86.00
1,500	5	417.68	11	12	93.31	86.13
2,000	1	784.98	14	16	93.06	84.05
2,000	2	969.60	14	16	93.36	85.65
2,000	3	855.88	14	16	93.12	85.20
2,000	4	1,037.95	14	16	92.64	83.60
2,000	5	794.92	14	16	92.68	84.40

Table A.2. Generated Instances: Class 2

No. of items	Instance no.	CPU (s)	Lower bound	No. of bins	Support (%)	Items supported (%)
50	1	0.24	1	1	85.76	66.00
50	2	0.15	1	1	90.30	64.00
50	3	0.21	1	1	92.86	74.00
50	4	0.25	1	1	87.48	68.00
50	5	0.08	1	1	91.10	66.00
100	1	1.32	1	1	81.91	66.00
100	2	0.94	1	1	86.11	74.00
100	3	1.63	1	1	87.08	73.00
100	4	2.03	1	1	91.75	77.00
100	5	2.08	1	1	87.36	70.00
150	1	13.80	2	2	92.37	76.00
150	2	13.51	1	2	90.78	74.67
150	3	14.43	1	2	91.17	74.67
150	4	25.25	1	2	92.01	76.00
150	5	10.20	1	2	93.67	76.00
200	1	49.60	2	2	89.62	80.50
200	2	56.44	2	2	93.68	82.50
200	3	46.87	2	2	92.94	81.50
200	4	88.03	2	2	91.96	79.00
200	5	35.49	2	2	89.85	75.50
500	1	148.14	4	5	92.63	80.00
500	2	207.27	4	5	92.17	81.20
500	3	149.97	4	5	91.14	78.40
500	4	101.81	4	5	91.09	80.00
500	5	114.67	4	5	91.79	78.60
1,000	1	266.68	7	9	92.34	83.40
1,000	2	273.50	7	9	92.20	82.70
1,000	3	303.37	7	9	92.78	84.70
1,000	4	190.73	7	9	91.67	82.10
1,000	5	189.00	7	9	93.18	83.40
1,500	1	504.79	11	12	93.04	85.13
1,500	2	570.13	11	12	92.96	84.47
1,500	3	508.58	11	12	93.51	86.13
1,500	4	793.34	11	12	92.77	84.80
1,500	5	550.26	11	12	92.59	84.20
2,000	1	1,187.42	14	16	92.71	84.30
2,000	2	983.60	14	16	92.86	85.65
2,000	3	915.78	14	16	92.71	85.05
2,000	4	941.06	14	16	92.97	85.50
2,000	5	1,147.59	14	16	93.06	85.05

Table A.3. Generated Instances: Class 3

No. of items	Instance no.	CPU (s)	Lower bound	No. of bins	Support (%)	Items supported (%)
50	1	0.13	1	1	82.99	58.00
50	2	0.15	1	1	85.53	64.00
50	3	0.30	1	1	94.85	70.00
50	4	0.14	1	1	90.68	64.00
50	5	0.12	1	1	90.39	58.00
100	1	1.64	1	1	83.29	70.00
100	2	1.30	1	1	90.97	78.00
100	3	1.71	1	1	86.15	68.00
100	4	2.09	1	1	90.64	74.00
100	5	0.76	1	1	86.26	68.00
150	1	29.15	1	2	89.93	80.67
150	2	13.76	1	1	90.67	78.00
150	3	18.79	1	2	88.86	80.00
150	4	17.55	1	2	88.32	77.33
150	5	25.78	1	2	88.94	79.33
200	1	66.92	2	2	91.52	77.00
200	2	44.47	2	2	92.82	80.50
200	3	98.57	2	2	89.97	77.00
200	4	76.35	2	2	92.03	77.00
200	5	66.14	2	2	90.24	76.50
500	1	181.33	3	4	92.73	83.00
500	2	416.39	3	4	91.22	79.60
500	3	252.67	3	4	91.76	80.40
500	4	410.61	3	4	91.04	79.20
500	5	462.00	3	4	92.04	81.80
1,000	1	378.94	6	7	92.16	84.60
1,000	2	389.29	6	7	92.09	84.80
1,000	3	355.97	6	7	92.46	83.60
1,000	4	658.97	6	7	91.61	84.20
1,000	5	338.88	6	7	92.29	85.00
1,500	1	1,032.15	9	11	92.21	83.60
1,500	2	801.09	9	11	92.30	83.33
1,500	3	750.07	9	10	92.04	85.07
1,500	4	780.79	9	10	92.84	84.93
1,500	5	1,059.09	9	11	92.28	83.60
2,000	1	1,330.08	12	14	91.89	83.35
2,000	2	1,068.59	12	14	92.94	85.35
2,000	3	1,243.27	12	14	92.53	85.10
2,000	4	1,103.31	12	14	92.71	84.75
2,000	5	1,295.56	12	14	92.45	84.95

Table A.4. Generated Instances: Class 4

No. of items	Instance no.	CPU (s)	Lower bound	No. of bins	Support (%)	Items supported (%)
50	1	0.14	1	1	90.20	62.00
50	2	0.10	1	1	88.06	50.00
50	3	0.18	1	1	90.98	62.00
50	4	0.18	1	1	87.48	62.00
50	5	0.06	1	1	87.28	60.00
100	1	1.82	1	1	88.91	75.00
100	2	1.47	1	1	86.08	71.00
100	3	1.89	1	1	86.65	74.00
100	4	1.39	1	1	89.13	72.00
100	5	2.30	1	1	88.61	77.00
150	1	25.24	1	2	91.82	80.67
150	2	37.13	1	1	88.41	76.67
150	3	10.17	1	1	88.58	78.00
150	4	16.41	1	1	90.90	80.67
150	5	19.38	1	1	88.81	78.67
200	1	49.42	2	2	88.42	73.50
200	2	77.96	1	2	91.60	76.50
200	3	46.22	2	2	90.71	74.50
200	4	113.91	2	2	90.26	75.00
200	5	88.18	2	2	90.66	76.50
500	1	340.42	3	4	91.64	81.80
500	2	261.55	3	4	91.58	82.00
500	3	167.61	3	4	90.59	78.60
500	4	242.38	3	4	91.53	80.00
500	5	252.75	3	4	91.97	80.60
1,000	1	516.74	6	7	90.78	82.20
1,000	2	706.60	6	7	91.21	81.40
1,000	3	470.70	6	7	91.93	82.80
1,000	4	395.40	6	7	91.48	82.20
1,000	5	592.44	6	7	90.45	81.70
1,500	1	1,074.27	8	10	91.70	82.87
1,500	2	838.53	8	10	90.93	82.67
1,500	3	1,099.02	8	10	92.72	83.53
1,500	4	886.32	8	10	91.48	82.80
1,500	5	823.89	8	10	91.73	83.40
2,000	1	1,458.63	11	12	92.72	85.80
2,000	2	1,145.12	11	13	92.16	84.20
2,000	3	1,442.41	11	13	92.24	85.10
2,000	4	1,813.35	11	13	92.04	83.60
2,000	5	1,451.93	11	13	91.85	84.55

Appendix B. Layer-Selection Algorithm

$\mathcal{L}$ and $\mathcal{SL}$ refer to the sets of generated and selected layers, respectively. The parameter de_l is the density of layer l, and it is calculated using the expression $\frac{\sum_{s \in \mathcal{S}_l} w_s d_s h_s}{WD o^l}$.

Algorithm B.1 (Layer-Selection Algorithm)

1: **procedure** INITIALIZATION
2:　Set $\mathcal{SL} = \emptyset$.
3: **end procedure**
4: **procedure** SELECT-LAYERS($\overline{\mathcal{L}}$)
5:　Sort layers in $\mathcal{SL}$ in descending order by decision variables α_l.
6:　**for** each layer $l \in \overline{\mathcal{L}}$ **do**
7:　　**for** each item i in layer l **do**
8:　　　**if** item i is already covered **then**
9:　　　　Go to next layer.
10:　　　**end if**
11:　　**end for**
12:　Set $\mathcal{SL} \leftarrow \mathcal{SL} \cup l$.
13:　**end for**
14: **end procedure**

Appendix C. Layer-Reorganization Algorithm

$\mathcal{SL}$ refers to the set of selected layers.

Algorithm C.1 (Layer-Reorganization Algorithm)

```
 1: procedure REPLACE-ITEMS(SL)
 2:     for each layer l ∈ SL do
 3:         SL ← SL \ l.
 4:         for each item i in l do
 5:             if i is covered before then
 6:                 Remove i from l.
 7:             end if
 8:         end for
 9:         Clear layer l and place the remaining items in it using Maxrects.
10:         Sort items not in l by λ_i in descending order.
11:         Use Maxrects to put the sorted items to l.
12:         SL ← SL ∪ l.
13:     end for
14: end procedure
```

Appendix D. Bin-Construction Algorithm

$\mathcal{B}$ refers to the set of constructed bins and $\mathcal{SL}$ represents the set of selected layers. Parameters h_b and h_l are the total height of the layers in the bin b and the height of layer l, respectively. H is the maximum height of a bin. Because of the sorting rule in line 5, the possibility that each item is vertically supported is increased. A lower bound to 3DBPP is given by $\lfloor \frac{v_{RMP}}{H} \rfloor$, if 3DSPP is solved by using column generation. The lower bounds L_0, L_1 and L_2 that are proposed by Martello et al. (2000) may also be used.

Algorithm D.1 (Bin-Construction Algorithm)

```
 1: procedure Initialization
 2:     Set B = ∅.
 3: end procedure
 4: procedure Construct–Bins(SL)
 5:     Sort layers in SL in descending order by density.
 6:     Open an empty bin b and set B ← B ∪ b.
 7:     for each layer l ∈ SL do
 8:         for each bin b ∈ B do
 9:             if h_b + h_l ≤ H then
10:                 Place l in b.
11:             end if
12:         end for
13:         if l cannot be placed in any bin b ∈ B then
14:             Open a new empty bin n and place l in n.
15:             Set B ← B ∪ n.
16:         end if
17:     end for
18: end procedure
```

Appendix E. S-Shaped Placement Algorithm

$\mathcal{N}$ refers to the set of items that are not covered. The parameter *itr* is the iteration count. To keep the overall height constant throughout the top sections of the bins, we alternate the placement of items in descending and ascending orders of height in each consecutive iteration. This reduces the negative impact on vertical stability.

Algorithm E.1 (The S-Shaped Placement Algorithm)
1: **procedure** PLACE-ITEMS($\mathcal{N}$)
2:     Sort $\mathcal{N}$ in descending order by width and depth.
3:     **for** each bin $b \in \mathcal{B}$ **do**
4:         $itr \leftarrow 1$.
5:         **while** $\mathcal{N} \neq \varnothing$ **do**
6:             **if** $itr \equiv 1 \pmod{2}$ **then**
7:                 Sort $\mathcal{N}$ in descending order by height.
8:             **else**
9:                 Sort $\mathcal{N}$ in ascending order by height.
10:            **end if**
11:            Place the items in an S-Shape.
12:            **if** no other item can be fit in current layer **then**
13:                Go up a layer, and set $itr \leftarrow itr + 1$.
14:            **end if**
15:            **if** no other item can be fit in current bin **then**
16:                Go to next bin.
17:            **end if**
18:         **end while**
19:     **end for**
20: **end procedure**

Appendix F. Superitem Generator

$\mathcal{SI}$ refers to the set of superitems, while $\mathcal{CI}$ refers the set of items from which superitems can be created. In line 20, $|i|$ is the number of items currently in superitem i and *MAX–ITEMS* is the maximum number of items that can be stacked on top of one another for a superitem. In our experiments, we used the value of 2 for this parameter. At the end of the algorithm, $\mathcal{CI}$ includes all the items and superitems that can be used to generate layers.

Algorithm F.1 (Superitem Generation Algorithm)
1: **procedure** Initialization
2:     Set $\mathcal{SI} = \varnothing$.
3:     Set $\mathcal{CI} = \varnothing$.
4: **end procedure**
5: **procedure** Generate Super–Items($\mathcal{I}$)
6:     **for** each set I_s of items with identical dimensions **do**
7:         **if** $|I_s| > 1$ **then**
8:             Create all possible superitems with 2 items by putting items side-by-side.
9:         **end if**
10:         **if** $|I_s| > 3$ **then**
11:             Create all possible superitems with 4 items by putting items side-by-side.
12:         **end if**
13:         Add the created superitems to $\mathcal{SI}$.
14:     **end for**
15:     Set $\mathcal{CI} \leftarrow \mathcal{I} \cup \mathcal{SI}$.
16:     Sort each item $i \in \mathcal{LI}$ in ascending order by width, depth, and height.
17:     **for** each item $i \in \mathcal{CI}$ **do**
18:         **for** each item $j > i \in \mathcal{CI}$ **do**
19:             **if** $w_j d_j \geq w_i d_i \geq 0.7 w_j d_j$ **then**
20:                Create superitem k by stacking item j over item i such that
21:                $x_i = \frac{w_j - w_i}{2}$ and $y_i = \frac{d_j - d_i}{2}$.
22:                Set $\mathcal{CI} \leftarrow \mathcal{CI} \cup k$.
23:                Set $\mathcal{SI} \leftarrow \mathcal{SI} \cup k$.
24:            **end if**
25:         **end for**
26:     **end for**
27: **end procedure**

Appendix G. Item-Replacement Algorithm

$\mathcal{NL}$ refers to the set of new layers obtained by replacing an item in a preexisting layer and $\mathcal{L}$ represents the set of generated layers.

Algorithm G.1 (Item-Replacement Algorithm)

```
 1: procedure Initialization
 2:     Set 𝒩ℒ = ∅.
 3: end procedure
 4: procedure Replace–Items(ℐ, ℒ)
 5:     for each layer l ∈ ℒ do
 6:         for each item i in layer l do
 7:             for each item j not in layer l do
 8:                 if 0.7w_i d_i ≤ w_j d_j ≤ w_i d_i then
 9:                     Replace i with j to create layer k.
10:                     Set 𝒩ℒ ← 𝒩ℒ ∪ k.
11:                 end if
12:             end for
13:         end for
14:     end for
15:     Set ℒ ← ℒ ∪ 𝒩ℒ.
16: end procedure
```

Appendix H. Item-Grouping Algorithm

Algorithm H.1 (Item-Grouping Algorithm)

```
 1: procedure Initialization
 2:     Set ℐ𝒢 = ∅.
 3: end procedure
 4: procedure GroupItems(ℐ)
 5:     Sort items in ℐ in descending order by height.
 6:     Create an item group IG_1 for the first item in ℐ, and set ℐ𝒢 ← ℐ𝒢 ∪ IG_1.
 7:     for each item i > 1 ∈ ℐ do
 8:         if h_i ≠ h_{i−1} then
 9:             Create an item group IG_i for item i, and set ℐ𝒢 ← ℐ𝒢 ∪ IG_i.
10:         end if
11:     end for
12:     for each item group g ∈ ℐ𝒢 do
13:         for each item i > g ∈ ℐ do
14:             if h_g − h_i ≤ 5 then
15:                 Set IG_g ← IG_g ∪ i.
16:             end if
17:         end for
18:     end for
19:     for each item group g ∈ 𝒢 do
20:         if |IG_g| = 1 then
21:             Set ℐ𝒢 ← ℐ𝒢 \ IG_g.
22:         end if
23:     end for
24: end procedure
```

Appendix I. Sample Figures for LCGA Results on Generated Instances

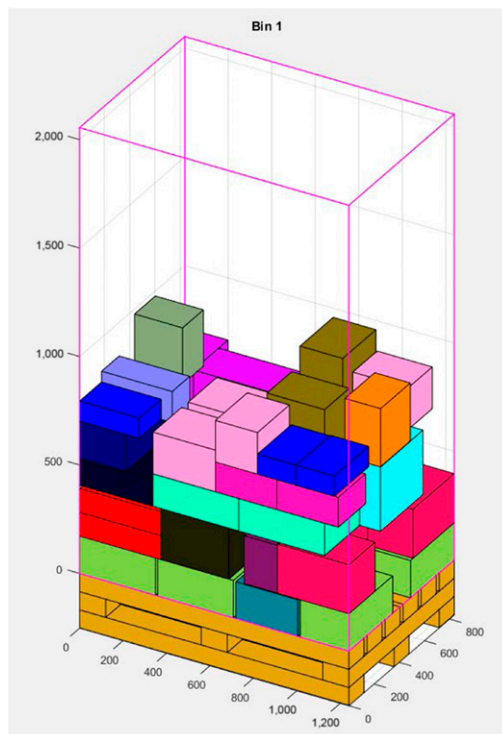**Figure I.1.** (Color online) Solution for Class 1, 50 Items, Instance 1

Figure I.2. (Color online) Solution for Class 2, 200 Items, Instance 1
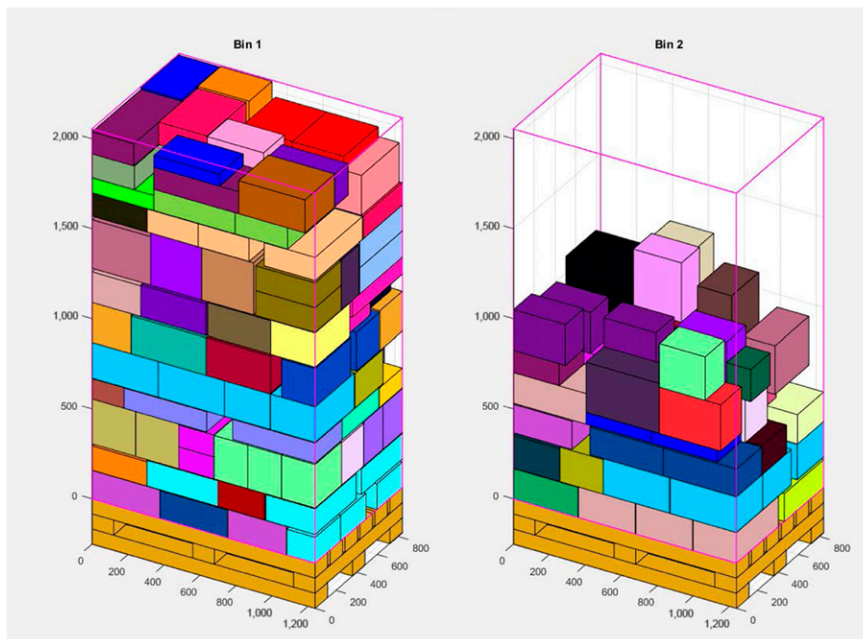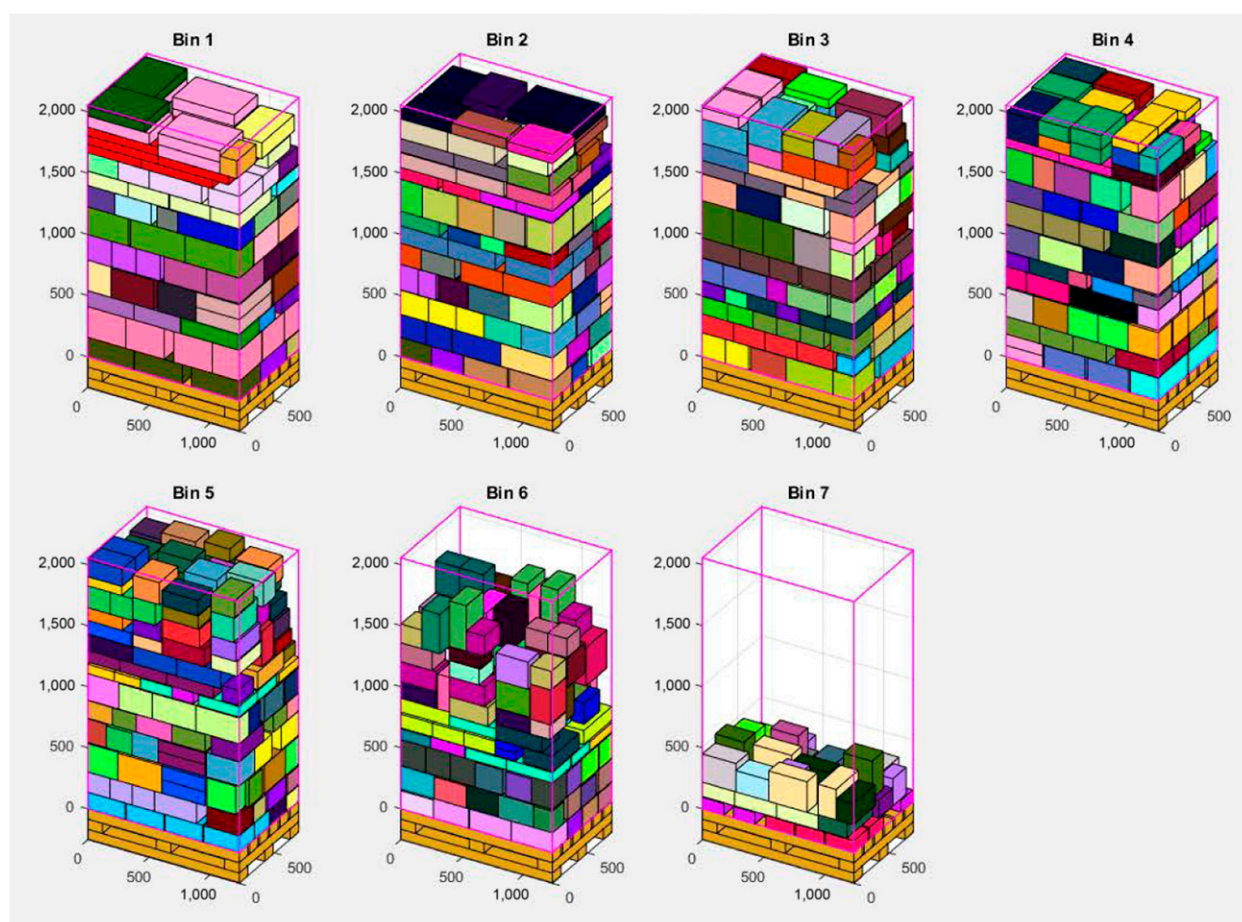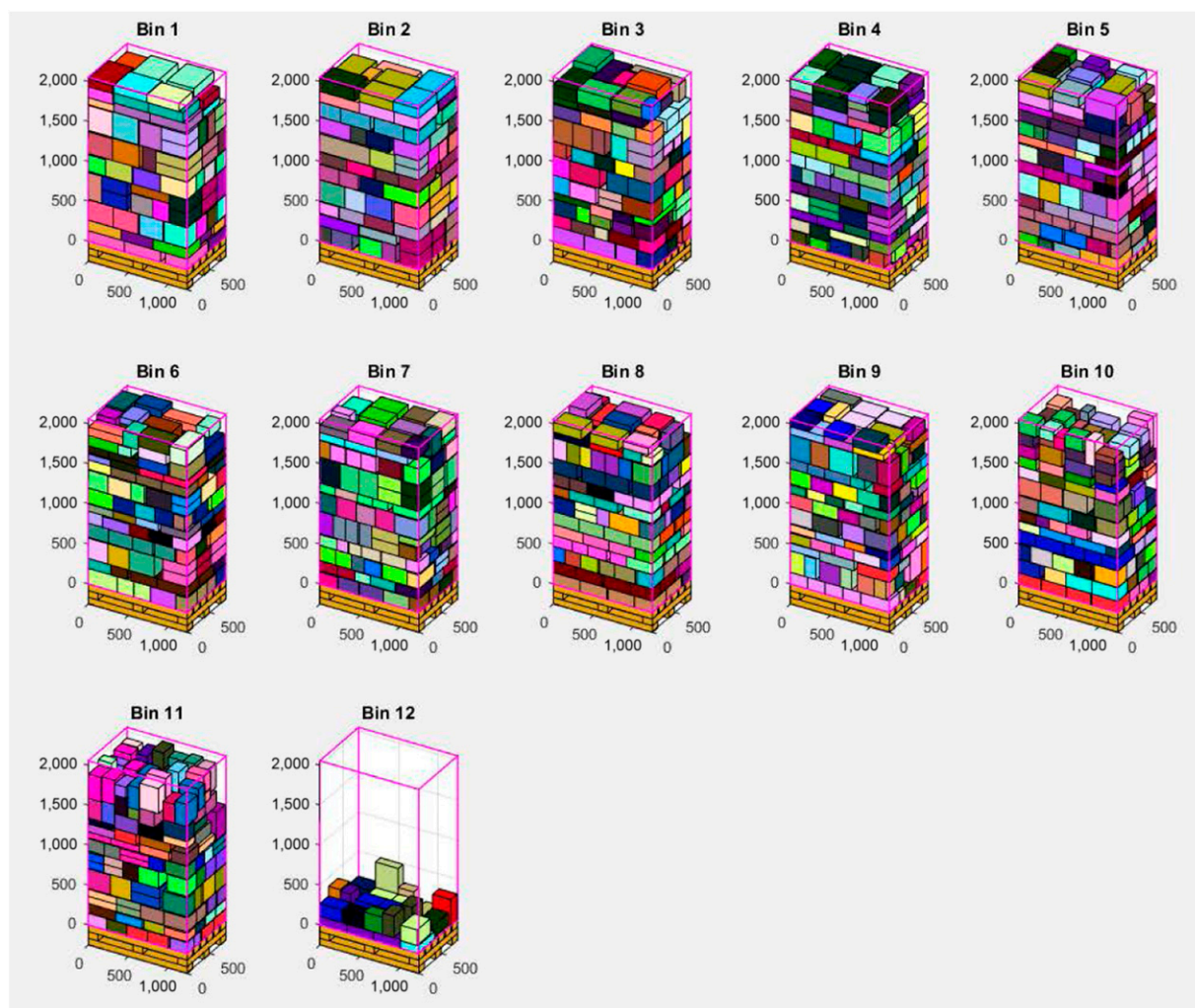
Figure I.3. (Color online) Solution for Class 3, 1,000 Items, Instance 1

Figure I.4. (Color online) Solution for Class 4, 2,000 Items, Instance 1

References

Bettinelli A, Ceselli A, Righini G (2008) A branch-and-price algorithm for the two-dimensional level strip packing problem. *4OR* 6(4):361–374.

Bischoff EE, Ratcliff M (1995) Issues in the development of approaches to container loading. *Omega* 23(4):377–390.

Bortfeldt A, Wäscher G (2013) Constraints in container loading – A state-of-the-art review. *Eur. J. Oper. Res.* 229(1):1–20.

Boschetti MA (2004) New lower bounds for the three-dimensional finite bin packing problem. *Discrete Appl. Math.* 140(1):241–258.

Chen C, Lee SM, Shen Q (1995) An analytical model for the container loading problem. *Eur. J. Oper. Res.* 80(1):68–76.

Clautiaux F, Nguyen A, Brenaut J (2015) Model for the challenge renault/ESICUP: Version 1.3. Accessed October 20, 2016, http://challenge-esicup-2015.org/doc/modele_renault.pdf.

Crainic TG, Perboli G, Tadei R (2008) Extreme point-based heuristics for three-dimensional bin packing. *INFORMS J. Comput.* 20(3):368–384.

Crainic TG, Perboli G, Tadei R (2009) TS^2PACK: A two-level Tabu search for the three-dimensional bin packing problem. *Eur. J. Oper. Res.* 195(3):744–760.

Cui YP, Zhou Y, Cui Y (2017) Triple-solution approach for the strip packing problem with two-staged patterns. *J. Combin. Optim.* 34(2):588–604.

Delignette-Muller ML, Dutang C (2015) Fitdistrplus: An *R* package for fitting distributions. *J. Statist. Software* 64(4):1–34.

Faroe O, Pisinger D, Zachariasen M (2003) Guided local search for the three-dimensional bin-packing problem. *INFORMS J. Comput.* 15(3):267–283.

Fekete SP, Schepers J, Van der Veen JC (2007) An exact algorithm for higher-dimensional orthogonal packing. *Oper. Res.* 55(3):569–587.

Hifi M, Kacem I, Nègre S, Wu L (2010) A linear programming approach for the three-dimensional bin-packing problem. *Electronic Notes Discrete Math.* 36:993–1000.

Junqueira L, Morabito R, Yamashita DS (2012) Three-dimensional container loading models with cargo stability and load bearing constraints. *Comput. Oper. Res.* 39(1):74–85.

Jylänki J (2010) A thousand ways to pack the bin—A practical approach to two-dimensional rectangle bin packing. Accessed March 10, 2016, http://clb.demon.fi/files/RectangleBinPack.pdf.

Lodi A, Martello S, Vigo D (2002) Heuristic algorithms for the three-dimensional bin packing problem. *Eur. J. Oper. Res.* 141(2):410–420.

Lodi A, Martello S, Vigo D (2004) Models and bounds for two-dimensional level packing problems. *J. Combin. Optim.* 8(3):363–379.

Martello S, Pisinger D, Vigo D (1998) Algorithm 864. Accessed March 30, 2017, http://www.diku.dk/pisinger/codes.html.

Martello S, Pisinger D, Vigo D (2000) The three-dimensional bin packing problem. *Oper. Res.* 48(2):256–267.

Martello S, Pisinger D, Vigo D, Boef ED, Korst J (2007) Algorithm 864: General and robot-packable variants of the three-dimensional bin packing problem. *ACM Trans. Math. Software* 33(1):Article No. 7.

Paquay C, Schyns M, Limbourg S (2016) A mixed integer programming formulation for the three-dimensional bin packing problem deriving from an air cargo application. *Internat. Trans. Oper. Res.* 23(1–2):187–213.

Parreño F, Alvarez-Valdés R, Oliveira J, Tamarit JM (2010) A hybrid GRASP/VND algorithm for two-and three-dimensional bin packing. *Ann. Oper. Res.* 179(1):203–220.

Ryan DM, Foster BA (1981) An integer programming approach to scheduling. Wren A, ed. *Computer Scheduling of Public Transport: Urban Passenger Vehicle and Crew Scheduling* (North-Holland, Amsterdam), 269–280.

Toffolo TAM, Esprit E, Wauters T, Berghe GV (2017) A two-dimensional heuristic decomposition approach to a three-dimensional multiple container loading problem. *Eur. J. Oper. Res.* 257(2):526–538.

Vance PH, Barnhart C, Johnson EL, Nemhauser GL (1994) Solving binary cutting stock problems by column generation and branch-and-bound. *Comput. Optim. Appl.* 3(2):111–130.

Wu Y, Li W, Goh M, de Souza R (2010) Three-dimensional bin packing problem with variable bin height. *Eur. J. Oper. Res.* 202(2):347–355.

Zhao X, Bennell JA, Bektaş T, Dowsland K (2016) A comparative review of 3D container loading algorithms. *Internat. Trans. Oper. Res.* 23(1–2):287–320.

Zhu W, Lim A (2012) A new iterative-doubling Greedy-Lookahead algorithm for the single container loading problem. *Eur. J. Oper. Res.* 222(3):408–417.

Zhu W, Huang W, Lim A (2012a) A prototype column generation strategy for the multiple container loading problem. *Eur. J. Oper. Res.* 223(1):27–39.

Zhu W, Zhang Z, Oon WC, Lim A (2012b) Space defragmentation for packing problems. *Eur. J. Oper. Res.* 222(3):452–463.