A Tetris Beam Search Algorithm for the Distributor's Pallet Loading Problem

Davide Croci[a,b,*], Ola Jabali[a,c], Jacopo Libe[b], Federico Malucelli[a], Joe Naoum-Sawaya[d]

[a]*Politecnico di Milano, Milan, Italy*
[b]*E80 Group S.p.a., Reggio Emilia, Italy*
[c]*HEC Montréal, Montréal, Canada*
[d]*Ivey Business School, University of Western Ontario, London, Canada*

Abstract

We consider the Distributor's Pallet Loading Problem (DPLP), where a set of cuboid-shaped items should be packed in identical pallets, satisfying several practical requirements. In particular, each item may be arranged in multiple orientations, must maintain static stability, and may withstand a limited weight. Furthermore, the combined weight of the items in each pallet must not exceed its total weight limit. We consider first minimizing the number of used pallets, and second maximizing their average pack density. DPLPs are commonly solved by layer-building methods, which generate and stack compact layers of items. Such methods are generally successful when considering a set of fairly homogeneous items. However, we consider a setting where highly heterogeneous items must be packed, for which we develop a beam search algorithm called Tetris Beam Search (TBS). This algorithm is based on a new constructive heuristic for the DPLP called Tetris Heuristic (TH). Inspired by the dynamics of the popular game Tetris, TH fills pallets by creating compact layers when possible, and non-compact structures when necessary. We evaluate TBS on generated test instances from the literature, where it significantly outperforms other competing methods. TBS reduces the average number of open bins by 22% and increases the average pack density by 15%. Notably, these improvements are realized while achieving more than 95% savings in average computational time. Finally, we present results evaluating the proposed algorithm's effectiveness on large real industrial instances obtained from an industrial partner.

Keywords: packing, distributor's pallet loading problem, 3D bin packing, beam search

1. Introduction

The e-commerce sector has been rapidly growing over the past years, accounting for 5.2 trillion U.S. dollars in 2021, and is expected to exceed 8 trillion dollars by 2026 (Chevalier, 2022). This rapid growth is accompanied by an increased demand for diverse products that are

[*]Corresponding author

available to consumers. However, the rise in product variety presents complex challenges to logistics operations, particularly in the storage and transportation of goods. Palletization, the process of stacking items onto pallets, is an integral part of these logistics operations, impacting costs, product safety, and overall efficiency. By automating and optimizing the arrangement of goods, warehouses may increase storage space utilization and streamline operations to handle the highly variable and increased demands of e-commerce. In this context of highly heterogeneous items, optimizing palletization emerges as an essential problem.

Motivated by the aforementioned reasons, this work focuses on the Distributor's Pallet Loading Problem (DPLP), which consists of stacking a set of cuboid-shaped items using identical three-dimensional pallets. The DPLP is an extension of the Three-Dimensional Bin Packing Problem (3DBPP), which includes additional constraints used to obtain pallet layouts that can be used in practical warehousing operations. In particular, in this paper, we consider (1) orientation constraints, ensuring that items are packed in all possible orthogonal orientations; (2) static stability constraints, guaranteeing that items will not fall due to gravity; (3) load-bearing constraints, implying that each item must sustain at most a given load of items placed on top of it; and (4) weight-limit constraints, indicating that a pallet's total weight does not exceed a given threshold. While standard 3DBPP only aims to minimize the number of open bins, real-world DPLP favors solutions where, for the same number of bins, the empty space inside each assembled structure is minimal. As proposed in Gzara et al. (2020), a proxy for measuring the volume of the empty space is the pack density, i.e., the ratio between the total volume of the items packed in a bin and the volume of the bin up to the highest item placed inside it. Maximizing the average pack density among all open bins thus minimizes empty spaces. Therefore, we consider a hierarchical set of objective functions in which we first minimize the number of open bins and then maximize the average pack density among all open bins.

The common practical approaches for solving real-world variants of the 3DBPP mainly rely on heuristic methods. Unlike exact methods, heuristics are favored because they produce effective solutions for large-scale problem instances in a timely manner. Many heuristic algorithms for 3DBPP employ a two-phased approach, where the first phase precomputes a series of compact structures by assembling items with a given loading pattern strategy, and the second phase packs these structures into one or more bins. Compact structures precomputed in the first phase are generally represented in the second phase as a single large object, delimited by the envelope cuboid of its internal items. Fanslau and Bortfeldt (2010) classified the loading pattern strategies into five major types: layer building, wall building, stack building, block building, and guillotine cutting. Recent approaches for the DPLP are mainly based on the layer-building strategy, where items with homogeneous heights are assembled into layers, that are then stacked atop one another inside each bin by solving a one-dimensional packing problem (Gzara et al., 2020; Dell'Amico and Magnani, 2021; Tresca et al., 2022). Other loading patterns

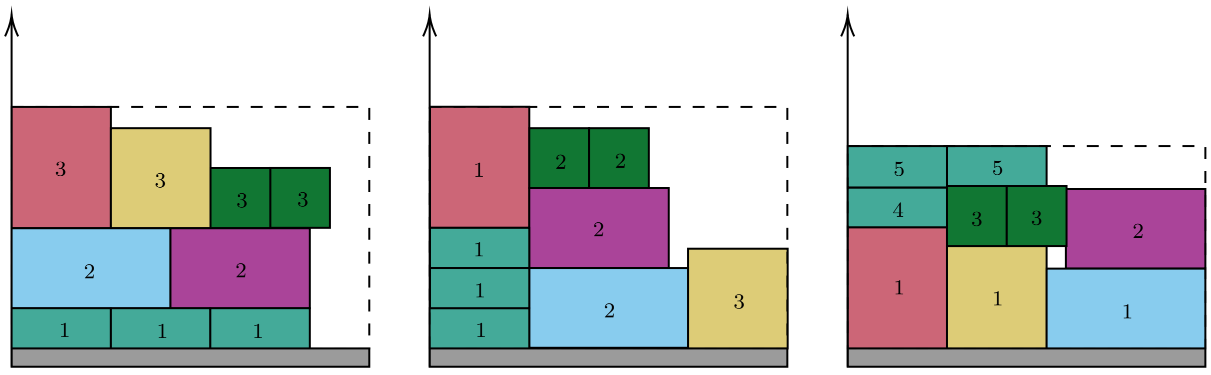


Figure 1: Lateral view of items packed using three different strategies, from left to right: Layer Building, Wall Building, and Tetris Heuristic. The number on each item denotes the order in which they are placed by the respective strategy.

have also been used to address 3DBPPs with practical constraints, such as wall building (Silva et al., 2020), stack building (Kır and Yazgan, 2019), and block building (Fanslau and Bortfeldt, 2010).

Layer-building methods are the most widely used algorithms to solve the DPLP because they implicitly favor the construction of stable structures even when stability is not explicitly enforced (Elhedhli et al., 2019). These methods are mostly successful when it is possible to generate compact layers with minimal empty spaces. They are thus effective when dealing with mostly homogeneous items, i.e., items with very similar dimensions, since this facilitates their combination in multiple efficient layers. However, in many real-world DPLPs, items are highly heterogeneous in terms of their dimensions, and precomputing efficient layers becomes very challenging.

To address the cases with highly heterogeneous items, we propose a new Tetris-based Beam Search (TBS) algorithm for the DPLP. TBS combines a Beam Search (BS) algorithm with a new constructive heuristic called Tetris Heuristic (TH). While standard layer-based methods aim to precompute compact layers that fill the bin's area as much as possible, TH works by filling a series of horizontal planes dynamically during the construction of a bin. This strategy allows us to create compact layers when possible while also obtaining non-compact structures that efficiently fill the bin's volume when necessary. To illustrate the difference between our approach and other loading strategies, Figure 1 compares TH to the commonly used layer and wall-building patterns.

While all the strategies result in the same number of bins, we observe that the layer and wall-building approaches produce more empty spaces despite generating layers and walls with high volume usage. TH, on the other hand, exploits the heterogeneity of items and combines them to obtain a structure with a higher pack density, i.e., with less empty space.

To the best of our knowledge, no other work deals with strongly heterogeneous DPLP with a similar constructive strategy where structures are not computed in a separate step. The main contributions of this paper are as follows:

- We develop Tetris Beam Search, a new beam search algorithm for the DPLP.
- We introduce a new constructive heuristic for the DPLP called Tetris Heuristic (TH), which acts as the core component for TBS.
- We introduce a series of algorithmic enhancements aimed at a fast evaluation of static stability and load-bearing constraints. In particular, we evaluate static stability constraints through bounding volume hierarchies, and we evaluate load-bearing constraints through a new matrix-based procedure.
- We demonstrate the effectiveness of our algorithm using comprehensive computational experiments on test instances from the literature. The results show that the proposed approach outperforms recent literature in both solution quality and computational efficiency. Furthermore, we demonstrate the effectiveness of our algorithm on a set of industrial instances, which we make publicly available.

The rest of this paper is organized as follows. In Section 2, we present a review of the relevant literature. In Section 3, we propose a mixed integer nonlinear formulation for the DPLP. In Section 4, we present our solution algorithm. In Section 5, we present computational experiments and lastly, in Section 6, we present our conclusions.

2. Literature Review

As a challenging problem with numerous applications, the Pallet Loading Problem (PLP) has received significant attention from the scientific community over the past decades. Two variants of the PLP have been mainly explored in the literature: the Manufacturer's Pallet Loading Problem (i.e., MPLP), and the Distributor's Pallet Loading Problem (DPLP). In the MPLP, a set of identical items are arranged into a bin with the objective of maximizing the total loaded volume. This problem is typically approached by solving a Two-Dimensional Bin Packing Problem (2DBPP), which identifies the optimal placement of items in a single layer. Subsequently, the formation of pallets is done by stacking these layers one atop another (Steudel, 1979; Morabito and Morales, 1998; Birgin et al., 2010; Silva et al., 2014). Recent efforts in the MPLP domain are mostly focused on the integration of practical constraints, such as static stability and items fragility (Gunawardena et al., 2021; Neuenfeldt Júnior et al., 2022). The DPLP, first introduced by Hodgson (1982), involves loading a set of heterogeneous items in the minimal number of bins. The DPLP is typically studied as an extension of the standard 3DBPP that includes practical constraints, such as static stability and load bearing (Sørensen et al., 2016; Ancora et al., 2020; Gzara et al., 2020; Tresca et al., 2022).

Beyond the domain of pallet loading, several other practical applications of 3DBPPs have also been studied in the literature. Notably, the Container Loading Problem (CLP) focuses on the placement of cuboid-shaped items within one or more shipping containers. The objective is to optimize space utilization while respecting weight and stability constraints (Bischoff and Ratcliff, 1995). Another important application is the Three-Dimensional Loading Capacitated Vehicle Routing Problem (3L-CVRP), which deals with the efficient distribution of goods by a fleet of vehicles to customers while accounting for the loading of the vehicles. The central focus of this latter dimension is the positioning of cuboid-shaped items in the vehicle, ensuring optimal space usage and satisfying logistical constraints (Gendreau et al., 2006; Ceschia et al., 2013). Regardless of the application, adapting the 3DBPP to real-world scenarios typically involves introducing new constraints into its standard structure, reflecting the complexities of practical logistics.

The problem that we consider in this paper is classified as a DPLP with orientation, static stability, load-bearing, and weight-limit constraints. While orientation and weight-limit constraints can be easily formulated, different models have been proposed for static stability and load-bearing constraints. Table 1 provides a summary of how these constraints have been modeled in the existing literature. For each work, we report the considered problem, its family (according to the typology of Wäscher et al., 2007), the technique used to model each practical constraint, and the proposed solution methodology. In the remainder of this section we review the literature summarized in Table 1. Specifically, in Sections 2.1 and 2.2, we review studies on modeling static stability and load bearing constraints, respectively. In Section 2.3, we review the different solution methodologies that have been proposed for the DPLP and other related problems.

Reference	**Problem**	**Family**	**Orientation**	**Static stability**	**Load Bearing**	**Solution method**
Junqueira et al. (2012)	CLP	SKP	ALL	AREA	WUA	MILP
Paquay et al. (2014)	CLP	MBSBPP	ALL	VERT	FRAG	MILP
Alonso et al. (2016)	PLP	SSSCSP	HOR	AREA	STACK	GRASP
Paquay et al. (2018)	CLP	MBSBPP	ALL	VERT	FRAG	Matheuristic
Silva et al. (2020)	CLP	SLOPP	ALL	AREA	FRAG	Matheuristic
Gzara et al. (2020)	DPLP	SBSBPP	HOR	AREA	LUA	Column Generation
Deplano et al. (2021)	CLP	MHKP	HOR	VERT	WOI	MILP
do Nascimento et al. (2021)	CLP	SKP	ALL	AREA	WUA	Exact method
Dell'Amico and Magnani (2021)	DPLP	SBSBPP	HOR	AREA	STACK	Matheuristic
Tresca et al. (2022)	DPLP	SBSBPP	HOR	AREA	STACK	Matheuristic
This work	DPLP	SBSBPP	ALL	AREA	LUA	Beam Search

- **Family**: single bin-size bin packing problem (SBSBPP), multiple bin-size bin packing problem (MBSBPP), single stock-size cutting stock problem (SSSCSP), single knapsack problem (SKP), single large object placement problem (SLOPP), multiple heterogeneous knapsack problem (MHKP).
- **Orientation**: horizontal only (HOR), every orientation (ALL).
- **Static Stability**: base area support (AREA), base vertices support (VERT), static mechanical equilibrium (MECH).
- **Load Bearing**: fragility (FRAG), stackability (STACK), maximum weight per unit area (WUA), maximum load per unit area (LUA), maximum weight of overlying items (WOI).
- **Solution method**: mixed-integer linear programming (MILP), greedy randomized adaptive search (GRASP)

Table 1: 3DBPPs with static stability, load bearing and rotation constraints.

2.1. Static stability

Static stability constraints ensure that items do not rotate or fall in the presence of gravity. Bortfeldt and Wäscher (2013) notes that static stability is among the most important constraints for practical 3DBPPs. A typical approach in the literature to address static stability is to enforce that a certain percentage of the base area of each item should lie on top of other items (Alonso et al., 2016; Silva et al., 2020; Gzara et al., 2020; do Nascimento et al., 2021; Dell'Amico and Magnani, 2021; Tresca et al., 2022). Alternatively, Ramos et al. (2016) study a CLP with static stability enforced through static mechanical equilibrium conditions. Since these conditions can be evaluated only when a bin is completely built, the authors also partially enforce static stability by imposing that the vertical projection of each item's center of mass must lie in the convex hull of the contact points between its base and the top face of items placed below it. Another line of research focuses on imposing conditions on the minimum number of base vertices for each item that must be supported (Paquay et al., 2014, 2018; Deplano et al., 2021). It is worth noting that guaranteeing a minimum of three supported vertices is sufficient to respect the partial condition of Ramos et al. (2016).

In this work, we propose a flexible algorithm that allows modeling static stability either with a minimal percentage of the base area to be supported, or with a minimum number of base vertices to be in contact with supporting surfaces. To compare with existing literature, we will use the former criterion in the remainder of this work.

2.2. Load bearing

Load bearing constraints ensure that items are not damaged by restricting how they can be placed on top of each other. One way to deal with load bearing constraints is by classifying items as fragile and non-fragile, and imposing that non-fragile items cannot be placed over fragile ones (Wang et al., 2010; Bortfeldt, 2012; Paquay et al., 2014, 2018). Similarly, load bearing is often modeled through the concept of stackability (Alonso et al., 2016; Tresca et al., 2022), which introduces an index for each item and imposes that items with lower indices cannot be stacked beneath those with higher indices. Another approach for imposing load bearing constraints sets a threshold on the weight of overlying items (Deplano et al., 2021). The load borne by an item is calculated by summing the weights of items placed above it, i.e., items whose center of mass' vertical projection intersects with the upper surface of the item itself. In the same spirit, other works impose an upper limit on the pressure that each item can endure (Dell'Amico and Magnani, 2021; do Nascimento et al., 2021). In these cases, the pressure exerted upon an item is determined by summing the ratio of weight to base area of items placed above it. Finally, Gzara et al. (2020) introduce a weight limit for each item, coupled with a graph representation for load computations.

We adopt the same load bearing model as Gzara et al. (2020), which enables more flexible item placement by not restricting certain items from being placed on top of others, as long as their maximum load capacity is not surpassed. Additionally, this model considers the influence of the weights of items that may not be in direct contact, offering a more realistic representation of load distribution. From Table 1, we observe that ours is the sole other study that uses this complex load-bearing model. Furthermore, from a computational perspective, we develop a new matrix-based procedure that efficiently computes the load borne by each item in the graph. Our procedure greatly reduces the computational time required to evaluate load-bearing constraints, as we will show in Section 5.

2.3. Solution methods

The 3DBPP is NP-hard (Bischoff and Marriott, 1990), and different approaches have been proposed in the literature for its solution. Wäscher et al. (2007) and Ali et al. (2022) provide extensive reviews on 3DBPPs. Due to the problem's complexity, exact algorithms can find optimal solutions only for small and simple problem instances, thus solution methods for real-world scaled problems predominantly rely on metaheuristic algorithms. In the context of standard 3DBPP, two of the best-performing algorithmic structures are the Biased Random Key Genetic Algorithm (BRKGA) proposed by Gonçalves and Resende (2012), and the Beam Search (BS) technique devised by Araya and Riff (2014).

Considering 3DBPP variants that include practical constraints, Paquay et al. (2014) and Deplano et al. (2021) developed MILP models for the Container Loading Problem. An exact algorithm for the same problem was developed by do Nascimento et al. (2021), where the CLP is decomposed into three subproblems with increasingly tight constraints. In particular, the proposed approach solves a one-dimensional knapsack problem to select the most voluminous subset of items to be packed in each bin, and the remaining models compute a feasible placement of items in each bin. Whenever the second or the third model does not find a feasible solution, a cut is added to eliminate the current solution, and the process is repeated. Using this method, the authors solved instances from the literature with up to 91 items in at most one hour of computational time.

Matheuristic algorithms have typically been applied to real-world 3DBPPs as decomposition-oriented methods coupled with a loading pattern strategy. Silva et al. (2020) propose a two-step wall-building algorithm, where a MILP model is first solved to generate an initial set of feasible item walls. The highest-volume walls are then placed within each bin by solving a second MILP model. A similar technique is used by Tresca et al. (2022), although in that case the first model generates a set of layers, while the second model stacks them in each bin.

Regarding metaheuristic algorithms, Ramos et al. (2016) developed a multi-population BRKGA for the CLP with static stability constraints. Their algorithm is based on a constructive heuristic that uses a maximal-spaces representation of empty spaces. Alonso et al. (2016) solved the problem of pallet loading and truck loading using a GRASP algorithm with a constructive phase to build feasible solutions, diversified with a randomized strategy. An improvement phase is then performed, where the moves involve swapping items or pallets between trucks. Lastly, Gzara et al. (2020) proposed a heuristic layer based column generation approach, where the columns are layers generated heuristically that are then stacked in bins.

We propose a Beam Search algorithm for the DPLP, inspired by its proven efficacy in solving standard 3DBPP instances (Araya and Riff, 2014). The main challenge lies in enforcing the static stability and load bearing constraints. Previous methods have leveraged loading pattern strategies to simplify the assessment of such constraints. Although effective for cases with mostly homogeneous items, these strategies falter in the presence of strongly heterogeneous items. To overcome this, we propose a new constructive heuristic for the DPLP called Tetris Heuristic (TH), which places items without using a specific loading pattern strategy.

3. Problem definition

In this section, we introduce a nonlinear programming formulation for the DPLP. As expected, solving this model with an optimization solver is computationally intensive and impractical even for small problem instances. As such, the purpose of the presentation of this model is to facilitate the description and understanding of the DPLP, while the computational results discussed in Section 5 are based on the Tetris Beam Search algorithm presented in Section 4. We first introduce the formulation without the static stability and load-bearing constraints in Sections 3.1 and 3.2, respectively. For ease of reference, all the notations are summarized in Appendix A.

To formulate the DPLP, we assume the following. Let I be the set of items to be packed in a set of identical bins B. We assume that the total number of bins, $|B|$, is sufficient to pack all items in I. Each item $i \in I$ is a rectangular parallelepiped (henceforth referred to as "cuboid") with integer width w_i, depth d_i, and height h_i. Each bin $b \in B$ is a cuboid of integer width W, depth D, and height H. Let μ_i be the weight of each item $i \in I$, and let M be the maximum weight that can be loaded in each bin. We assume that every bin has a coordinate system in which the x-axis, y-axis, and z-axis are aligned with the bin's width, depth, and height, respectively (see Figure 2). The origin $(0, 0, 0)$ of this coordinate system lies on the front-left-bottom corner of the bin. We consider the possibility of placing items in different orientations. In particular, let $O = \{wdh, whd, dwh, dhw, hwd, hdw\}$ be the ordered set of the six orthogonal orientations that each item can assume in a three-dimensional space. These orientations correspond to aligning the edges of the item with each of the Cartesian axes. For example, whd denotes the orientation in which an item's width, depth, and height are respectively aligned to the x-axis, z-axis, and y-axis of the Cartesian coordinate system. For the sake of simplicity, we will refer to each orientation with an integer number from 0 to 5, in the same order as per the definition of O. To account for real-world scenarios where some items can only be oriented in specific ways, we define $O_i \subseteq O$ as the set of feasible orientations

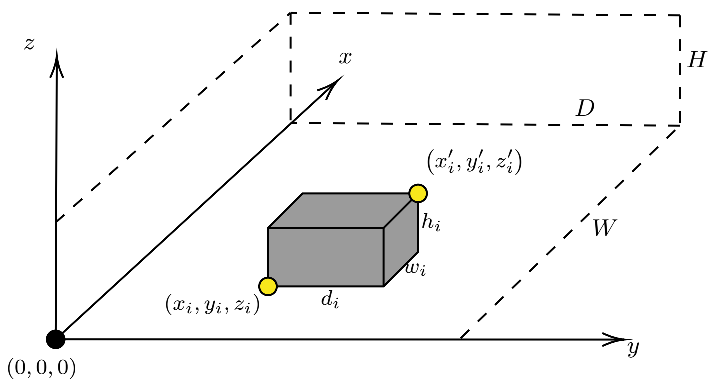


Figure 2: Representation of the system of coordinates of a bin with an item.

for item $i \in I$.

As the dimensions of the items are all integer values, we define the following integer decision variables

$(x_i, y_i, z_i) \in \mathbb{Z}^+$	Cartesian coordinates of the front-left-bottom corner of item i,	$\forall i \in I$,
$(x'_i, y'_i, z'_i) \in \mathbb{Z}^+$	Cartesian coordinates of the rear-right-top corner of item i,	$\forall i \in I$,
$z_b^{max} \in \mathbb{Z}^+$	the maximum vertical coordinate reached by the items placed in bin b,	$\forall b \in B$.

We also define the following binary variables

$$u_b = \begin{cases} 1 & \text{if bin } b \in B \text{ is used,} \\ 0 & \text{otherwise} \end{cases} \qquad \forall b \in B;$$

$$v_{ib} = \begin{cases} 1 & \text{if item } i \in I \text{ is assigned to bin } b \in B, \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in I, b \in B;$$

$$\rho_{ij}^x = \begin{cases} 1 & \text{if item } i \text{ is behind item } j, \text{ i.e., } x_i \geq x'_j, \\ 0 & \text{otherwise} \end{cases} \qquad \forall i, j \in I;$$

$$\rho_{ij}^y = \begin{cases} 1 & \text{if item } i \text{ is on the right of item } j, \text{ i.e., } y_i \geq y'_j, \\ 0 & \text{otherwise} \end{cases} \qquad \forall i, j \in I;$$

$$\rho_{ij}^z = \begin{cases} 1 & \text{if item } i \text{ is above item } j, \text{ i.e., } z_i \geq z'_j, \\ 0 & \text{otherwise} \end{cases} \qquad \forall i, j \in I;$$

$$r_{oi} = \begin{cases} 1 & \text{if item } i \text{ is placed with orientation } o, \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in I, o \in O_i.$$

We formulate the DPLP as

$$\min \quad \sum_{b \in B} u_b \tag{1}$$

$$
\begin{aligned}
\max \quad & \sum_{b \in B} \frac{\sum_{i \in I} w_i d_i h_i v_{ib}}{W D z_b^{max}} & & (2)\\
\text{s.t.} \quad & v_{ib} \le u_b & \forall i \in I, \forall b \in B \quad & (3)\\
& \sum_{b \in B} v_{ib} = 1 & \forall i \in I \quad & (4)\\
& \sum_{i \in I} \mu_i v_{ib} \le M & \forall b \in B \quad & (5)\\
& \sum_{o \in O_i} r_{oi} = 1 & \forall i \in I \quad & (6)\\
& x'_i - x_i = w_i(r_{0i} + r_{1i}) + d_i(r_{2i} + r_{3i}) + h_i(r_{4i} + r_{5i}) & \forall i \in I \quad & (7)\\
& y'_i - y_i = w_i(r_{2i} + r_{4i}) + d_i(r_{0i} + r_{5i}) + h_i(r_{1i} + r_{3i}) & \forall i \in I \quad & (8)\\
& z'_i - z_i = w_i(r_{3i} + r_{5i}) + d_i(r_{1i} + r_{4i}) + h_i(r_{0i} + r_{2i}) & \forall i \in I \quad & (9)\\
& z_b^{max} \ge z'_i - H(1 - v_{ib}) & \forall i \in I, b \in B \quad & (10)\\
& \rho_{ij}^x + \rho_{ji}^x + \rho_{ij}^y + \rho_{ji}^y + \rho_{ij}^z + \rho_{ji}^z \ge v_{ib} + v_{jb} - 1 & \forall (i,j) \in I, b \in B \quad & (11)\\
& x'_k \le x_i + W(1 - \rho_{ik}^x) & \forall (i,k) \in I \mid i \ne k \quad & (12)\\
& x_i + 1 \le x'_k + W\rho_{ik}^x & \forall (i,k) \in I \mid i \ne k \quad & (13)\\
& y'_k \le y_i + D(1 - \rho_{ik}^y) & \forall (i,k) \in I \mid i \ne k \quad & (14)\\
& y_i + 1 \le y'_k + D\rho_{ik}^y & \forall (i,k) \in I \mid i \ne k \quad & (15)\\
& z'_k \le z_i + H(1 - \rho_{ik}^z) & \forall (i,k) \in I \mid i \ne k \quad & (16)\\
& x_i, x'_i \in [0, W] \subset \mathbb{Z}^+ & \forall i \in I \quad & (17)\\
& y_i, y'_i \in [0, D] \subset \mathbb{Z}^+ & \forall i \in I \quad & (18)\\
& z_i, z'_i, z_b^{max} \in [0, H] \subset \mathbb{Z}^+ & \forall i \in I, b \in B \quad & (19)\\
& u_b, v_{ib}, r_{oi}, \rho_{ik}^x, \rho_{ik}^y, \rho_{ik}^z \in \{0,1\} & \forall i,k \in I, b \in B, o \in O_i. \quad & (20)
\end{aligned}
$$

The objective function (1) minimizes the number of used bins, while the objective function (2) maximizes the sum of pack densities of all bins. These objectives are considered in lexicographic order: firstly, we minimize the number of bins, and secondly, we maximize the sum of pack densities. Constraints (3) ensure that each item is assigned to an open bin, while constraints (4) force each item to be assigned to exactly one bin. Constraints (5) ensure that the weight limit is respected for each bin. Constraints (6) force each item to have exactly one orientation. Constraints (7), (8), and (9) set the values of x'_i, y'_i, and z'_i, respectively, for each item $i \in I$, accounting for its rotation. Constraints (10) set the value of the maximum height z_b^{max} reached by items in each bin $b \in B$. Constraints (11) impose that two items assigned to the same bin must not overlap. In other words, considering two items assigned to the same bin, one of them must be either behind, beside, or above the other. Constraints (12)–(13), (14)–(15), and (16) set the precedence variables along the x-axis, y-axis, and z-axis, respectively. Finally, constraints (17)–(20) set the domain of all variables. We note that by imposing variables x'_i, y'_i, and z'_i to be respectively smaller than W, D, and H, we forbid violations of the bin's maximum dimensions.

3.1. Static stability

Static stability constraints ensure that all the items packed in a bin will not topple due to gravity. An item j supports item i if its maximum vertical coordinate equals the minimum vertical coordinate of i, i.e., $z'_j = z_i$, and if the projections of the bases of both items on the XY-plane overlap. As typically seen in the literature (Alonso et al., 2016; Silva et al., 2020; Gzara et al., 2020; do Nascimento et al., 2021; Dell'Amico and Magnani, 2021; Tresca et al., 2022), we define an item as statically stable if the sum of the overlap area over the XY-plane with all of its supporting items is greater or equal to α percent of its base area. Figure 3 shows an example of a statically stable item. Similarly to Gzara et al. (2020) and Tresca et al. (2022), we introduce a "tolerance" β on the vertical gap between supporting items to account for measurement errors. Thus, item j can support item i if $z_i - \beta \leq z'_j \leq z_i$.

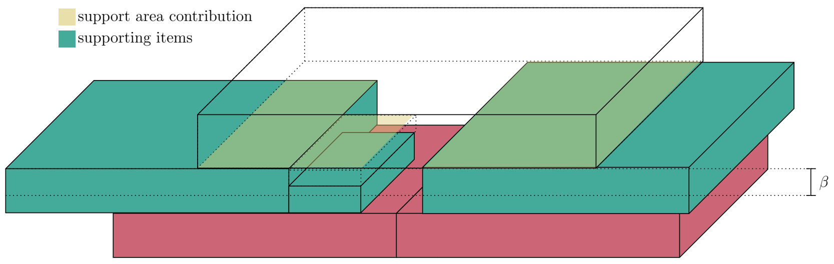


Figure 3: Example of a transparent item supported by three green items. The item is considered statically stable if the support area contribution is greater than α% of its base area.

Static stability is enforced through a set of nonlinear constraints that are added to problem (1)–(20). To formulate the static stability constraints, we define the following integer variables

$$s^x_{ij} = \text{ length of the overlap segment (if exists) between items } i \text{ and } j \text{ along the } x\text{-axis} \quad \forall i, j \in I,$$
$$s^y_{ij} = \text{ length of the overlap segment (if exists) between items } i \text{ and } j \text{ along the } y\text{-axis} \quad \forall i, j \in I;$$

and the following binary decision variables

$$z^c_{ij} = \begin{cases} 1 & \text{if item } i \in I \text{ may directly support item } j \in I, \\ 0 & \text{otherwise;} \end{cases} \qquad \forall i, j \in I$$

$$g_i = \begin{cases} 1 & \text{if item } i \in I \text{ lies on the ground,} \\ 0 & \text{otherwise;} \end{cases} \qquad \forall i \in I$$

$$t^b_{ij} = \begin{cases} 1 & \text{if items } i, j \in I \text{ are both placed in bin } b \in B, \\ 0 & \text{otherwise.} \end{cases} \qquad \forall i, j \in I, \forall b \in B$$

The static stability constraints are formulated as

$$s^x_{ij} = \max\{0, \min\{x'_i, x'_j\} - \max\{x_i, x_j\}\} \qquad \forall i, j \in I \qquad (21)$$
$$s^y_{ij} = \max\{0, \min\{y'_i, y'_j\} - \max\{x_i, x_j\}\} \qquad \forall i, j \in I \qquad (22)$$

$$z_i \leq H(1-g_i) \qquad \forall i \in I \tag{23}$$

$$z_j - z'_i \leq \beta + H(1-z^c_{ij}) \qquad \forall i,j \in I : i \neq j \tag{24}$$

$$z_j - z'_i \geq -H(1-z^c_{ij}) \qquad \forall i,j \in I : i \neq j \tag{25}$$

$$t^b_{ij} \geq v_{ib} + v_{jb} - 1 \qquad \forall i,j \in I, \forall b \in B \tag{26}$$

$$t^b_{ij} \leq v_{ib} \qquad \forall i,j \in I, \forall b \in B \tag{27}$$

$$t^b_{ij} \leq v_{jb} \qquad \forall i,j \in I, \forall b \in B \tag{28}$$

$$\sum_{j \in I : i \neq j} s^x_{ij} s^y_{ij} z^c_{ij} (\sum_{b \in B} t^b_{ij}) \geq \alpha (x'_i - x_i)(y'_i - x_i)(1-g_i) \qquad \forall i \in I \tag{29}$$

$$s^x_{ij}, s^y_{ij} \in \mathbb{Z}^+ \qquad \forall i,j \in I \tag{30}$$

$$z^c_{ij}, t^b_{ij}, g_i \in \{0,1\} \qquad \forall i,j \in I, b \in B. \tag{31}$$

Nonlinear constraints (21) and (22) set the values of s^x_{ij}, s^y_{ij}, i.e., the length of the overlap segment between items i and j. Constraints (23) set the value of g_i for each $i \in I$. Constraints (24) and (25) set the value of z^c_{ij} for each $i, j \in I$. Constraints (26), (27), and (28) set the value of t^b_{ij} for each $i, j \in I$ and $b \in B$. Finally, nonlinear constraints (29) impose static stability to each item $i \in I$ by making sure that at least α percent of its base area overlaps with the area of the other supporting items in the same bin.

3.2. Load bearing

Load-bearing constraints impose that the total weight supported by each item $i \in I$ is less than or equal to a threshold m_i, i.e., the maximum load supported by item i. To formulate these constraints we assume, as proposed by Gzara et al. (2020), that the weight of an item is distributed to the supporting items below based on the proportion of the base area they support. This means that if multiple items support the base of an item, each bears a fraction of the weight corresponding to the effective proportion of the area it covers. This assumption also applies in cases where items are not fully supported, as allowed by the static stability constraints. For example, if item i has 40% of its base area supported by item j and another 40% supported by item k, it will distribute its load half on j and half on k. Considering a set of items I_b packed in bin $b \in B$, we model the distribution of weights through a load-bearing graph, where each node represents an item $i \in I_b$ and a directed arc (i, j) exists if item $i \in I_b$ is supported by item $j \in I_b$. Figure 4 provides an example of a load-bearing graph. Every arc has a value a_{ij}, which represents the percentage of supported base area of item i specifically supported by item j. We introduce real variables l_{ij} to express the weight exerted by item i on item j, when placed directly above it. Real variables q_i represent the total weight loaded by other items on i, including those not directly placed above it. Load-bearing constraints are formulated as

$$l_{ij} = (1-g_i)(q_i + \mu_i) \frac{s^x_{ij} s^y_{ij} z^c_{ji} \sum_{b \in B} t^b_{ij}}{\sum_{k \in I \setminus \{i\}} s^x_{ik} s^y_{ik} z^c_{ki} \sum_{b \in B} t^b_{ik}} \qquad \forall i,j \in I : i \neq j \tag{32}$$

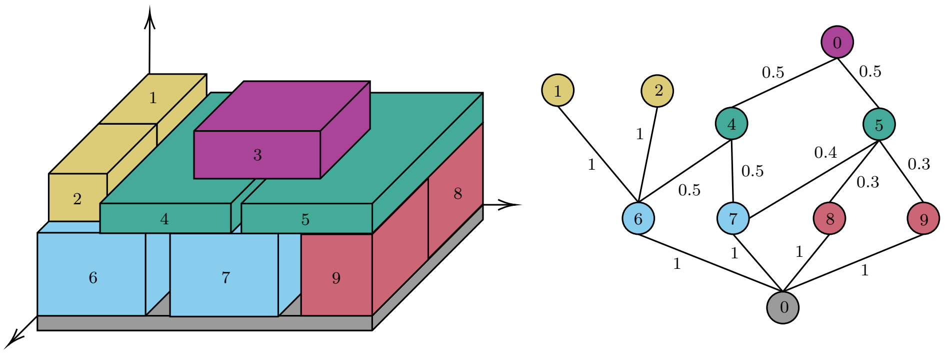


Figure 4: Example of a load-bearing graph for a set of items loaded in a bin.

$$q_i = \sum_{j \in I} l_{ji} \qquad \forall i, j \in I, \forall b \in B \tag{33}$$

$$q_i \leq m_i \qquad \forall i \in I \tag{34}$$

$$l_{ij}, q_i \in \mathbb{R}^+ \qquad \forall i, j \in I. \tag{35}$$

Constraints (32) set the value of l_{ij} for each pair of items $i, j \neq i \in I$. Constraints (33) set the load q_i borne by each item $i \in I$. Finally, constraints (34) ensure that the load supported by each item $i \in I$ does not exceed m_i.

Since solving the DPLP to optimality given formulation (1)–(35) is computationally prohibitive even for small-sized instances, next we present our Tetris Beam Search algorithm for the DPLP.

4. Solution algorithm

In this section, we introduce the Tetris Beam Search (TBS) algorithm, which is designed to solve the DPLP. TBS combines a tailored Beam Search algorithm with a new constructive heuristic, the Tetris Heuristic (TH), that we designed for the single-bin version of the DPLP. We start by introducing TH in Section 4.1, followed by the full TBS algorithm in Section 4.2.

4.1. Tetris Heuristic

The Tetris Heuristic (TH) is a constructive heuristic for the single-bin version of the DPLP (sDPLP) and is inspired by the dynamics of the popular game Tetris (Pažitnov, 1984). Given a set of items I to pack, each with its own set of feasible orientations $O_i \subseteq O$, and a single bin of dimensions (W, D, H) that can support a maximum weight M, the sDPLP consists of packing the most voluminous subset of items in the bin while respecting orientation, static stability, load-bearing, and weight limit constraints.

TH finds a feasible solution to the sDPLP by iteratively packing items from I in the bin, which involves fixing their coordinates and orientation. As items are introduced in the bin, they "fall" to the lowest available position, mirroring the mechanics of the Tetris game where geometric shapes descend to fill gaps at the bottom of the playing field. Specifically, the algorithm is based on the concept of “horizontal planes”, which are horizontal sections of the bin corresponding to the top surfaces of one or more packed items. Each horizontal plane p is defined by its vertical coordinate z_p. Throughout its execution, TH maintains the set P of all horizontal planes that may offer support for packing additional items. At each iteration, the algorithm selects the lowest plane p^* from P, packs a subset of items on p^*, removes p^* from P, and finally updates the remaining planes given the newly packed items. The process continues until P is empty or all items in I are packed.

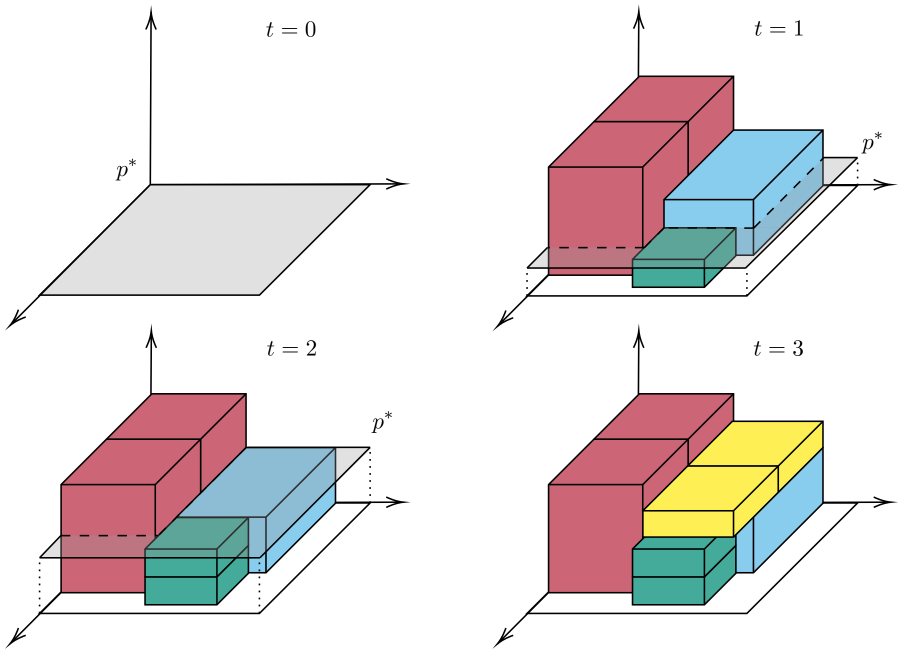


Figure 5: View of a bin built with TH. The lowest plane available at each iteration t is denoted with p^*.

Algorithm 1: Tetris Heuristic

```
  input  : $b, I$
  output: $\Gamma$
1 $P \leftarrow \{p_0\}$
2 $\Gamma \leftarrow \varnothing$
3 $R \leftarrow \{(i, o) \mid i \in I, o \in O_i\}$
4 while $|P| > 0 \wedge |R| > 0$ do
5     $p^* \leftarrow argmin_{p \in P}\, z_p$
6     $\Gamma_{p^*} \leftarrow CPH(R, p^*)$ // see Algorithm 3
7     $P \leftarrow UpdatePlanes(p^*, P, \Gamma_{p^*}, \Gamma)$ // see Algorithm 2
8     $\Gamma \leftarrow \Gamma \cup \Gamma_{p^*}$
9     $R \leftarrow \{(i, o) \in R \mid i \notin \Gamma_{p^*}\}$
```

Figure 5 shows an example of a bin built with TH, while the complete pseudocode of the procedure is shown in Algorithm 1. The procedure begins at line 1 by initializing the set P with a single plane p_0, corresponding to the bottom of the bin which has vertical coordinate $z_{p_0} = 0$.

Afterward, it initializes the empty set Γ, which holds all details about the packed items and serves as the output of TH. Specifically, Γ includes tuples (i, o_i, x_i, y_i, z_i), each representing an item $i \in I$ packed in the bin with orientation $o_i \in O_i$ at coordinates (x_i, y_i, z_i). Concluding the initialization phase, TH builds set R, which contains tuples (i, o) representing all combinations of items $i \in I$ and their feasible orientations $o \in O_i$. We refer with $w(i, o)$, $d(i, o)$, and $h(i, o)$, respectively, to the width, depth, and height of item i with orientation o, defined as follows

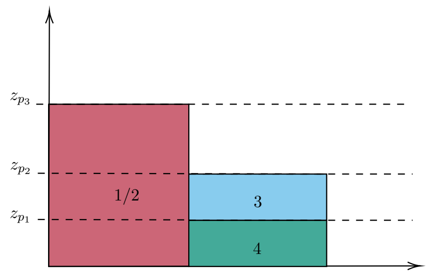
$$w(i,o) = \begin{cases} w_i & \text{if } o \in \{wdh, whd\}, \\ d_i & \text{if } o \in \{dwh, dhw\}, \\ h_i & \text{otherwise} \end{cases} \qquad d(i,o) = \begin{cases} w_i & \text{if } o \in \{dwh, hwd\}, \\ d_i & \text{if } o \in \{wdh, hdw\}, \\ h_i & \text{otherwise} \end{cases}$$

$$h(i,o) = \begin{cases} w_i & \text{if } o \in \{dhw, hdw\}, \\ d_i & \text{if } o \in \{whd, hwd\}, \\ h_i & \text{otherwise.} \end{cases}$$

The algorithm's main loop starts at line 4, iterating until either P is empty or all the items are packed. First, TH selects the plane $p^* \in P$ with the minimum vertical coordinate z_{p^*} at line 5. Second, at line 6, TH packs a subset of items Γ_{p^*} on p^* by solving a modified Two-Dimensional Orthogonal Knapsack Problem (2DOKP) with a tailored algorithm called Candidate Points-based Heuristic (CPH), which we describe in Section 4.1.1. Finally, the algorithm updates the set of horizontal planes P through the *UpdatePlanes* procedure, adds the newly packed items Γ_{p^*} to the output set Γ, removes all tuples from R that contain items that are already packed, and proceeds to the next iteration.

The core step of TH is the CPH procedure, which is used to select items and pack them on p^*. For this procedure to be executed, TH maintains two sets of items for each plane $p \in P$, denoted by L_p and T_p. Set $L_p \subseteq \Gamma$ includes the "supporting items", which are the items packed directly under plane p that may provide support for additional items to be placed above it. Set $T_p \subseteq \Gamma$ contains the "obstacle items", which are items packed with a maximum vertical coordinate greater than z_p. Items in L_p are used by the CPH procedure to enforce static stability and load-bearing constraints, while items in T_p are used to enforce non-overlap constraints. The usage of both sets significantly improves the performance of TH, as it restricts the number of items that need to be considered to evaluate the feasibility of each placement. Considering the example in Figure 5, the set P of horizontal planes for the second iteration is shown in Figure 6, together with the corresponding sets L_p and T_p for each plane $p \in P$.

The *UpdatePlanes* procedure updates the set of horizontal planes P at the end of each iteration. Its pseudocode is presented in Algorithm 2. Following the placement of items Γ_{p^*} on the lowest plane p^*, TH first removes p^* from P (line 1). Then, for each item $i \in \Gamma_{p^*}$, the algorithm checks if there is a plane $p' \in P$ that is at the same vertical coordinate as the



p	L_p	T_p
p_1	$\{4\}$	$\{1,2,3\}$
p_2	$\{3\}$	$\{1,2\}$
p_3	$\{1,2\}$	$\varnothing$

Figure 6: Horizontal planes for TH's $t = 1$ iteration from Figure 5.

top surface of i. If such a plane exists, the algorithm adds i to its supporting items $L_{p'}$ (line 5). Otherwise, a new plane p' is created at the top surface of i, and added to P (lines 7–11). Finally, the algorithm adds each packed item $i \in \Gamma_{p^*}$ to the obstacle items T_p of each plane $p \in P$ that lies below the top surface of i.

Algorithm 2: Update horizontal planes

```
   input  : $p^*, P, \Gamma_{p^*}, \Gamma$
   output: $P'$
1  $P' \leftarrow P \setminus \{p^*\}$
2  for $(i, o_i, x_i, y_i, z_i) \in \Gamma_{p^*}$ do
3      if $\exists\, p \in P \mid z_p = z_i + h(i, o_i)$ then
4          $p' \leftarrow p \in P \mid z_p = z_i + h(i, o_i)$
5          $L_{p'} \leftarrow L_{p'} \cup \{(i, o_i, x_i, y_i, z_i)\}$
6      else
7          $p' \leftarrow$ new horizontal plane
8          $z_{p'} \leftarrow z_i + h(i, o_i)$
9          $L_{p'} \leftarrow \{(i, o_i, x_i, y_i, z_i)\}$
10         $T_{p'} \leftarrow \{(\gamma, o_\gamma, x_\gamma, y_\gamma, z_\gamma) \in \Gamma \mid z_\gamma + h(\gamma, o_\gamma) > z_i + h(i, o_i)\}$
11         $P' \leftarrow P' \cup \{p'\}$
12 for $(i, o_i, x_i, y_i, z_i) \in \Gamma_{p^*}$ do
13     for $p \in P' \mid z_p < z_i + h(i, o_i)$ do
14         $T_p \leftarrow T_p \cup \{(i, o_i, x_i, y_i, z_i)\}$
```

4.1.1. Two-dimensional orthogonal knapsack problem

TH places items on plane p^* by solving a 2DOKP, which consists of packing the most valuable set of rectangular items in a rectangular bin. A comprehensive review of the literature on the 2DOKP can be found in Iori et al. (2022) and Cacchiani et al. (2022). We consider a rectangular bin of size (W, D), corresponding to the base of the bin in the sDPLP. We consider rectangular items for all combinations of items to pack and their feasible orientations $(i, o) \in R$, each with a size corresponding to the rotated item's base dimensions $(w(i, o), d(i, o))$. Each rectangle has value $v(i, o) = w_i d_i h_i$. This value implicitly accounts for the third dimension $h(i, o)$. The goal is to pack a subset of rectangles corresponding to distinct items that maximize the total packed value while ensuring no two rectangles overlap. Each rectangle $(w(i, o), d(i, o))$ packed at coordinates (x, y) on plane p^* corresponds to a three-dimensional placement of item

i with orientation o at coordinates (x, y, z_{p^*}).

Differently from the classical 2DOKP, in TH we must produce placements that account for orientation, static stability, load-bearing, and weight limit constraints of the DPLP, as well as non-overlap constraints with items in T_p. To address the intrinsic non-linearity of static stability and load-bearing constraints, we develop a new heuristic algorithm for the 2DOKP called Candidate points-based heuristic (CPH), which is detailed in Algorithm 3. The basic idea

Algorithm 3: Candidate points-based heuristic

```
   input  : $R, p^*$
   output: $\Gamma_{p^*}$
1  $\Gamma_{p^*} \leftarrow \emptyset$
2  $E \leftarrow$ initial candidate points
3  sort $R$ by decreasing volume $v(i,o)$ and break ties by area $w(i,o)d(i,o)$
4  forall $(i,o) \in R$ do
5      sort $E$ by increasing distance from origin $(0,0)$
6      forall $(x_e, y_e) \in E$ do
7          if $isFeasible(i, o, x_e, y_e, z_{p^*})$ then
8              $\Gamma_{p^*} \leftarrow \Gamma_{p^*} \cup \{(i, o, x_e, y_e, z_{p^*})\}$
9              $E \leftarrow E \setminus \{(x_e, y_e)\}$
10             $E \leftarrow E \cup \{(x_e + w(i,o), y_e), (x_e, y_e + d(i,o))\}$
11             $R \leftarrow \{(j,o) \in R \mid j \neq i\}$
12             break
```

behind CPH is that rectangles are placed in the bin by positioning their bottom-left corner on a point selected among a set of candidate points E, which the algorithm maintains throughout its execution. Whenever a rectangle (i.e., an item) is placed on a selected point, it generates new points that are added to the set of candidates. Specifically, the CPH procedure initializes the set E of candidate points with the origin $(0,0)$, the coordinates of the bottom-left corner of each supporting item in L_p, and the coordinates of the bottom-right and top-left corners of each obstacle item in T_p, i.e.,

$$\begin{aligned} E =& \{(0,0)\} \cup \{(x_i, y_i) \mid (i, o_i, x_i, y_i, z_i) \in L_p\} \cup \\ & \{(x_i + w(i,o_i), y_i), (x_i, y_i + d(i,o_i)) \mid (i, o_i, x_i, y_i, z_i) \in T_p\}. \end{aligned}$$

These points are included in E to favor the placement of items in R in feasible positions, i.e., positions that respect static stability and load-bearing constraints and do not overlap with items in T_p. After initialization, at line 3, the algorithm sorts all rectangles in R in decreasing order of value $v(i,o)$, breaking ties by decreasing area $w(i,o) \times d(i,o)$. For each rectangle $(i,o) \in R$, CPH sorts the candidate points in E by increasing distance from the origin $(0,0)$, and selects the first that is feasible for packing. The feasibility check is performed at line 7 by the $isFeasible$ procedure, which evaluates all constraints of the DPLP. This procedure, whose implementation is detailed in Section 4.3, is the bottleneck in the time complexity of CPH as it involves checking non-linear constraints for each tentative placement. When a rectangle (i, o) is placed on point $(x_e, y_e) \in E$, that point is removed from E, new candidate points $(x_e + w(i, o), y_e)$ and $(x_e, y_e + d(i, o))$ are added to E, and all other rectangles derived from the item i are removed from R. The algorithm then proceeds to the next iteration, terminating when all remaining rectangles in R have been evaluated.

CPH is inspired by the Extreme Point heuristic proposed by Crainic et al. (2008), which is designed for the classical 3DBPP. The main difference between the two algorithms is that CPH does not project non-dominated points on the opposite axis, but rather considers them directly as candidate insertion points. Furthermore, the initialization of the candidate points set E is improved to incorporate positions that are feasible with respect to static stability and load-bearing constraints of the DPLP.

4.2. Tetris Beam Search

Beam Search is a tree-search heuristic algorithm that explores the k most promising nodes at each level of the search tree (Lowerre and Reddy, 1976). In general, Beam Search explores the discrete space of feasible solutions (states) of a given problem to find the state that optimizes the problem's objective function. Each state represents either a partial or a complete solution to the considered problem, where a partial solution is one where not all variables have value. States are linked in a tree structure where each state is connected to its "children" by a defined transition operation, which usually involves fixing the value of one or more variables according to the parent node. In many state space search algorithms, the state space is too large to generate and store in memory. Therefore, it is usually represented implicitly, i.e., states are generated as they are explored and typically discarded when needed. Beam Search navigates the state space with a breadth-first strategy, where only the k most promising states are explored at each iteration, and the rest are discarded.

An outline of our Tetris Beam Search (TBS) for the DPLP is reported in Algorithm 4. Each state s is defined as a tuple (B_s, Γ_s, P_s), where $B_s \subseteq B$ is the set of open bins, Γ_s is the set of packed items, and P_s is the set of all horizontal planes in each open bin. Let Γ_b^s be the set of items packed in bin $b \in B_s$. Then, we define Γ_s as the union of the sets Γ_b^s for each open bin, i.e., $\Gamma_s = \bigcup_{b \in B_s} \Gamma_b^s$. Similarly, let P_b^s be the set of horizontal planes in each open bin $b \in B_s$. Set P_s is defined as the union of sets P_b^s, i.e., $P_s = \bigcup_{b \in B_s} P_b^s$. A state s is "final" if all items are packed, i.e., $|\Gamma_s| = |I|$. TBS receives as input the initial state $s_{in} = (\{b\}, \{\}, \{p_0\})$, where b is a single bin and p_0 is its initial plane, and gives as output s_{out}, which is the best final state found during the search. The algorithm begins at lines 1–2 by initializing the set of states to be processed S, and the set of final states F. The core of TBS is the loop at lines 4–11, which is performed as long as there are states in S. At each iteration, the algorithm applies the *ExpandState* procedure to each state $s \in S$ (line 6), generating its set of children states U_s. If U_s is empty, then s is a final state and is added to F. Otherwise, TBS stores

Algorithm 4: Beam search algorithm

```
input  : $s_{in}$
output: $s_{out}$
1  $S \leftarrow \{s_{in}\}$
2  $F \leftarrow \varnothing$
3  while $S \neq \varnothing$ do
4      $S' \leftarrow \varnothing$
5      forall $s \in S$ do
6          $U_s \leftarrow \mathit{ExpandState}(s)$ (see Alg. 5)
7          if $U_s = \varnothing$ then
8              $F \leftarrow F \cup s$
9          else
10             $S' \leftarrow S' \cup U_s$
11     $S \leftarrow \mathit{SelectStates}(S', k)$ (see Sect. 4.2.2)
12 $s_{out} \leftarrow \mathit{SelectStates}(F, 1)$ (see Sect. 4.2.2)
```

all the new children states in a set $S' = \bigcup_s^S U_s$. After all the states in S are processed, the algorithm selects the k most promising states from S' (line 11) to form the new set S, discards the remaining states, and proceeds to the next iteration. When S is empty, TBS terminates its main loop and selects the best final state from F as the output s_{out} (line 12).

4.2.1. State expansion

The *ExpandState* procedure, outlined in Algorithm 5, plays a crucial role in TBS as it is responsible for computing the set of children states U_s for a single input state s. The procedure begins at line 1 by generating the set R of all combinations of items to pack $i \in I \setminus \Gamma_s$ and their feasible orientations $o \in O_i$. This set R is then partitioned into a set of subsets G, where each $R_g \subseteq R$ groups together all items that share the same rotated height $h(i, o)$. Then, for each partition $g \in G$ and each open bin $b \in B_s$, the procedure generates a new state by placing a set of items Γ_{bg} in b (lines 7–11). The items in Γ_{bg} are determined by executing a single iteration of the Tetris Heuristic on b. Specifically, the lowest plane in b is selected at line 7, and the knapsack problem is solved at line 8 with only rectangles from R_g as input. If at least one item was packed, the planes in b are updated at line 10, and the new child state is added to U_s. For each partition $g \in G$, if no children states were found, a new empty bin b' is opened (line 13), and a child state is generated by placing items from R_g in the newly opened bin (lines 15–17).

4.2.2. State selection

The *SelectStates* procedure takes as input a set of states S and a parameter k and returns a subset of S containing its k best states according to specific ranking criteria. This procedure is called twice in Algorithm 4: at the end of each iteration (line 11), to select the states to be expanded in the next iteration, and at the end of the algorithm (line 12), to select the best state among all final states. In the latter case, the algorithm ranks final states using the two-level objective function of our DPLP, as defined in Section 3. In particular, for each state

Algorithm 5: ExpandState

```
   input  : $s = (B_s, \Gamma_s, P_s), I$
   output: $U_s$
1  $R \leftarrow \{(i, o) \mid i \in I \setminus \Gamma_s, o \in O_i\}$
2  $G \leftarrow$ partition $R$ by rotated height $h(i, o)$
3  $U_s \leftarrow \emptyset$
4  forall $g \in G$ do
5      $U_s^g \leftarrow \varnothing$
6      forall $b \in B_s$ do
7          $p^* \leftarrow \underset{p \in P_b^s}{argmin}\, z_p$
8          $\Gamma_{gb} \leftarrow CPH(R_g, p^*)$
9          if $\Gamma_{gb} \neq \emptyset$ then
10             $P_{gb} \leftarrow UpdatePlanes(p^*, P_b^s, \Gamma_{gb}, \Gamma_b^s)$
11             $U_s^g \leftarrow U_s^g \cup \{(B_s, \Gamma_s \cup \Gamma_{gb}, P_s \cup P_{gb})\}$
12     if $|U_s^g| = 0$ then
13         $b' \leftarrow New\ empty\ bin$
14         $P_{b'} \leftarrow \{p_0\}$
15         $\Gamma_{gb'} \leftarrow CPH(R_g, p_0)$
16         $P_{b'} \leftarrow UpdatePlanes(p_0, P_{b'}, \Gamma_{gb'}, \varnothing)$
17         $U_s^g \leftarrow U_s^g \cup \{(B_s \cup \{b'\}, \Gamma_s \cup \Gamma_{gb'}, P_s \cup P_{gb})\}$
18     $U_s \leftarrow U_s \cup U_s^g$
19 return $U_s$
```

$s = (B_s, \Gamma_s, P_s) \in F$ we compute the number of open bins $f(s) = |B_s|$, and the average pack density of open bins $g(s) = \frac{1}{|B_s|} \sum_{b \in B_s} \frac{\sum_{i \in \Gamma_b^s} w_i d_i h_i}{W D z_b^{max}}$. The state with the smallest $f(s)$ is selected, using the largest $g(s)$ as a tiebreaker. The selection of the most promising k states in S at the end of each iteration (line 11 of Algorithm 4) is more complex, as it involves predicting the final objective function value attainable from a search subtree rooted in each state $s \in S$. Let $f'(s)$ and $g'(s)$ be, respectively, the predicted minimal number of open bins and the predicted maximal average pack density of s. At intermediate stages of the algorithm, the *SelectStates* procedure uses $f'(s)$ and $g'(s)$ to select the best k states in S. We define $f'(s) = f(s) + f''(s)$ as the sum of the current number of open bins, plus an estimate $f''(s)$ of the total number of bins that must be opened to accommodate the items that have not been packed yet. Specifically, let $V_s = \sum_{i \in I \setminus \Gamma_s} w_i d_i h_i$ be the total volume of items to pack when starting from state s. Let $\hat{V}_s^b$ be an estimate of the volume of items that may still be packed in an open bin $b \in B_s$. We define $\hat{V}_s$ as $\sum_{b \in B_s} \hat{V}_s^b$, and compute $f''(s)$ as the lower bound on the number of additional bins needed to accommodate the unpacked items, i.e.,

$$f''(s) = \left\lceil \frac{V_s - \hat{V}_s}{WDH} \right\rceil . \tag{36}$$

Let $\hat{d}_i = \frac{\mu_i}{w_i d_i h_i}$ be the density of item $i \in I$, and let $\hat{d}_s = \frac{1}{|I \setminus \Gamma_s|} \sum_{i \in I \setminus \Gamma_s} \hat{d}_i$ be the average density of the items to pack in state s. Let $\phi_i = m_i - q_i$ be the residual load-bearing capacity of each item $i \in \Gamma_s$, and let $\phi_b^{min} = \min_{i \in \Gamma_b^s}\{\phi_i\}$ be the minimum residual load-bearing capacity between all items placed in bin $b \in B_s$. The maximal volume available for packing additional items in bin b is estimated as

$$\hat{V}_s^b = \min\{\frac{\phi_b^{min}}{\hat{d}_s}, WDH - \sum_{i \in I_b} w_i d_i h_i\}, \tag{37}$$

where the first term approximates the volume that can be packed without violating load-bearing constraints, and the second term ensures the bin's total volume is not exceeded.

To predict the maximal average pack density $g'(s)$, we consider the average smoothed pack density of state s. Specifically, let s_p be the parent state of s in the search tree, and let $g(s_p)$ be the average pack density of s_p. The average smoothed pack density of s is $g'(s) = SmoothCD(g(s_p), g(s))$, where $SmoothCD$ is the critically damped spring smoothing function described in Kirmse (2004). The implementation of $SmoothCD$ is described in Appendix B. The term "smoothing"refers to a technique wherein a value is gradually changed over time towards a desired target. In our context, we apply smoothing to the average pack density to reduce the significance of undesired and sudden changes when transitioning from one state to another. These fluctuations often occur when packing items into a new bin or on top of a structure with an already high pack density. Smoothing removes these sudden fluctuations, preventing unnecessary penalties for states that might still be favorable.

4.3. Constraints evaluation enhancements

In this section, we present algorithmic enhancements that improve the performance of the $isFeasible$ function, introduced in Section 4.1.1, which evaluates the feasibility of each tentative item placement. Since this procedure is called many times by CPH during the execution of TBS, improving its efficiency is crucial to reduce the overall computational time of the algorithm. Next, in Section 4.3.1, we describe the use of bounding volume hierarchies to speed up the constraints evaluation. Then, in Section 4.3.2, we describe a new matrix-based approach to evaluate the load-bearing constraints.

4.3.1. Bounding volume hierarchies

Bounding volume hierarchies (BVHs) are tree structures used to organize objects in a two or three-dimensional space. Their primary goal is to optimize various queries, such as collision detection or object intersection. At its core, a BVH consists of a balanced binary tree in which each node corresponds to a region of space, its child nodes partition that space into subvolumes, and leaf nodes are the actual objects.

In TBS, the evaluation of feasibility requires performing many object intersection queries. Specifically, the evaluation of non-overlap constraints requires checking for the three-dimensional intersection of item i, placed with orientation $o \in O_i$ on a plane p at coordinates (x_i, y_i, z_p), with all items in T_p. Moreover, the evaluation of static stability and load-bearing constraints for the same placement requires computing the overlap area on the XY-plane between i and its supporting items, which consists of many two-dimensional intersection queries between i and all items in L_p. A simple implementation of these checks would require $O(n)$ time, where n is the number of items in T_p or L_p. With the goal of reducing this complexity, we use BVHs which allow for intersection queries between items in $O(log(n))$ time. In particular, TBS maintains two BVHs for each plane p: one for T_p and one for L_p, and both trees are updated whenever these sets are updated.

4.3.2. Matrix-based load bearing constraints evaluation

As described in Section 3.2, the evaluation of load-bearing constraints requires traversing the load-bearing graph for each tentative placement. A straightforward implementation of this procedure may perform a BFS traversal of the graph with a complexity of $O(n + e)$ where n is the number of nodes in the graph and e is the number of edges. We propose an alternative matrix-based implementation that has a worst-case complexity of $O(n^2)$ but is much faster in practice due to its ability to exploit the Single Instruction Multiple Data (SIMD) operations of modern CPUs.

For each state s of TBS and each open bin $b \in B_s$, we maintain a matrix $C_b = \{c_{ij}\}_{i,j \in \Gamma_b^s}$. Each entry $c_{ij} \in [0, 1] \subset \mathbb{R}$ represents the percentage of item i's load q_i that is exerted on item j, as if it was computed by traversing the load-bearing graph. For example, consider two packed items i and j with a coefficient $c_{ij} = 0.5$. Placing an item k with weight μ_k entirely on top of item i results in a load increase of $0.5\mu_k$ kg on item j, even if i is not directly supported by j. Consistently, we have $c_{ii} = 1$ for all items $i \in \Gamma_b^s$.

Suppose we are evaluating the placement of a new item k in bin b, where k is not yet included in Γ_b^s. Let $M_b = \{m_i\}_{i \in \Gamma_b^s}$ be the array of maximum loads of items in Γ_b^s, and let $Q_b = \{q_i\}_{i \in \Gamma_b^s}$ be the array of current loads supported by items in Γ_b^s. To evaluate the load-bearing feasibility of placing item k, we first compute the array A_k of the direct contact area percentages between k and all items in Γ_b^s. Then, we evaluate the load-bearing constraints by checking the following inequality

$$Q_b + C_b(\mu_k A_k) \leq M_b. \tag{38}$$

The complexity of this check is $O(n^2)$, as it requires the multiplication of the $n \times n$ matrix C_b with the $n \times 1$ matrix $w_k A_k$. Although this complexity is higher than the $O(n + e)$ complexity of a BFS traversal, the matrix-based approach is much faster in practice due to its ability to exploit SIMD instructions. This allows for the parallel computation of multiple matrix entries, which significantly enhances computational efficiency.

All matrices C_b are initialized with a single entry $c_{00} = 1$, corresponding to the bin's base, and are updated whenever a new item is placed in the bin. Specifically, whenever item k is placed in bin b, we update C_b by adding a new row and column to it, as follows

$$C_b = \begin{pmatrix} C_b & C_b A_k \\ 0 & 1 \end{pmatrix}. \tag{39}$$

5. Computational results

In this section, we present comprehensive computational experiments evaluating the efficiency of TBS under various configurations and benchmarking its performance against existing methodologies from the literature. TBS was implemented in Java 8, and all experiments were conducted on a Linux machine equipped with an Intel Core i7-1165G7 CPU clocked at 2.80 GHz and with 16 GB of RAM.

Following the experimental framework proposed by Gzara et al. (2020), we used their instance generator to create 140 instances that are compatible with the ones presented in their paper. Specifically, each instance consists of a set of items I to be packed in multiple identical bins with dimensions $W = 1240$, $D = 840$, $H = 2200$, and the maximum weight that can be loaded in each bin is $M = 1500$. For each item $i \in I$, we are given its dimensions (w_i, d_i, h_i), weight μ_i, and load-bearing capacity m_i. All items can be rotated by 90 degrees around their vertical axis, therefore, the set of feasible orientations for each item i is $O_i = \{wdh, dwh\}$. A tolerance $\beta = 10$ is considered to determine the support surface between adjacent items, and a minimum support area of $\alpha = 0.7$ is required for static stability. We note that the instances that are generated using the Gzara et al. (2020) generator exhibit a high degree of item heterogeneity. For the 140 instances that we generated, on average, 2.17 items per instance have the same dimensions (w_i, d_i, h_i), weight μ_i, and load-bearing capacity m_i. All generated instances and their corresponding solutions are available online[1].

The remainder of this section is organized as follows. In Section 5.1, we evaluate the performance of TBS for different values of the beam width k to assess the trade-off between solution quality and computational time. In Section 5.2, we demonstrate the added value of the algorithmic enhancements introduced in Section 4.3. In Section 5.3, we compare the performance of TBS with other DPLP algorithms from the literature. Lastly, in Section 5.4, we provide results on a new set of instances obtained from our industrial partner, which we also make publicly available for future benchmarking purposes.

5.1. Beam width selection

The beam width k, which determines the number of states to be expanded at each iteration, is a crucial parameter to tune in the beam search algorithm. Large values of k allow for a more thorough exploration of the search space but also increase the algorithm's computational time.

[1]https://github.com/Daddeee/tetris-beam-search-instances

In this section, we evaluate the impact of different values of k on the performance of TBS. Table 2 reports the results obtained by TBS on the 140 generated instances, with values of k equal to 1, 5, 10, 20, 50, and 100. Each row reports the average number of open bins, pack density, and computational time considering the twenty instances with $|I|$ items. Detailed results are reported in Appendix C. We observe that the average number of open bins and the average pack density improve as k increases, while the average computational time grows. When k increases from 1 to 5, the average number of open bins decreases by 6.1%, and the average pack density increases by 7.4%. These improvements become less pronounced as k further increases. For example, when k grows from 50 to 100, the average number of open bins decreases only by 0.6%, and the average pack density increases by 1.3%. The computational time, on the other hand, grows linearly as k increases. In the remainder of this Section, we consider $k = 10$, as it provides a good compromise between solution quality and computational efficiency.

	Bins (#)						Pack Density (%)						Time (s)					
$\|I\| \setminus k$	1	5	10	20	50	100	1	5	10	20	50	100	1	5	10	20	50	100
100	1.10	1.00	1.00	1.00	1.00	1.00	0.60	0.68	0.69	0.71	0.75	0.75	0.09	0.21	0.60	0.80	1.79	3.94
150	2.00	1.70	1.65	1.55	1.50	1.50	0.55	0.63	0.66	0.69	0.72	0.73	0.11	0.50	1.01	2.21	4.18	10.81
200	2.00	2.00	2.00	2.00	2.00	2.00	0.65	0.69	0.69	0.71	0.71	0.72	0.19	0.88	1.43	3.15	7.93	13.95
500	4.50	3.80	3.70	3.60	3.65	3.50	0.65	0.74	0.75	0.78	0.78	0.81	1.01	3.56	5.72	11.46	29.02	48.20
1000	7.25	7.10	7.05	7.00	7.00	7.00	0.75	0.79	0.79	0.80	0.80	0.81	3.09	10.91	18.38	37.06	78.52	153.08
1500	10.75	10.15	9.90	9.80	9.75	9.75	0.77	0.80	0.82	0.83	0.84	0.84	5.63	21.47	35.20	71.74	146.58	315.65
2000	13.90	13.25	13.25	13.10	13.05	12.95	0.78	0.82	0.83	0.84	0.84	0.85	9.82	36.70	57.92	120.39	274.18	550.17
Average	5.93	5.57	5.51	5.44	5.42	5.39	0.68	0.73	0.75	0.77	0.78	0.79	2.85	10.60	17.18	35.26	77.46	156.54

Table 2: Results for different k on generated instances of Gzara et al. (2020).

5.2. *Algorithmic enhancements evaluation*

In Section 4.3, we introduced two algorithmic enhancements to improve the efficiency of TBS. In particular, we described the use of bounding volume hierarchies to speed up the evaluation of the non-overlap, stability, and load-bearing constraints. We also described a new matrix-based representation to speed up the evaluation of the load-bearing constraints.

To assess the impact of these algorithmic enhancements, we compare the performance of TBS with and without them on the generated instances. Table 3 reports the average computational time of TBS on these instances, while the complete results are reported in Appendix C. As the solution quality is not impacted by the algorithmic enhancement, we do not include the objective function values. Column "No BVH" reports the average computational time of TBS without using bounding volume hierarchies. Column "BFS Load" reports the average computational time of TBS where the load-bearing constraints are evaluated with a BFS traversal of the load-bearing graph, as proposed by Gzara et al. (2020). Lastly, column TBS reports the average computational time obtained by TBS with all the algorithmic enhancements introduced in Section 4.3. We note that in the "No BVH" column, we continue to use a matrix-based approach for the load-bearing evaluations, and in the "BFS Load" column, we continue to use BVH to compute item overlap. We observe that BVH improves the average computational time

$\lvert I \rvert$	No BVH	BFS Load	TBS
100	0.69	0.92	0.60
150	2.36	3.21	1.01
200	3.77	5.47	1.43
500	15.71	19.21	5.72
1000	55.18	47.67	18.38
1500	114.26	86.09	35.20
2000	202.20	129.89	57.92
Average	56.31	41.78	17.18

Table 3: Computational time (s) of TBS with $k = 10$ on generated instances of Gzara et al. (2020), with and without algorithmic enhancements.

by 69.5%, while using the matrix-based representation to evaluate the load-bearing constraints improves the average computational time by 58.9%. These results showcase the effectiveness of both algorithmic enhancements. Moreover, the observed higher impact of BVH on computational time shows that computing items overlap is an extremely time-consuming part of the algorithm.

5.3. Comparison with methods from the literature

We compare the performance of TBS to the approaches of Gzara et al. (2020) and Tresca et al. (2022). In the remainder of this section, we refer to the algorithm proposed by Gzara et al. (2020) as GEY and to the algorithm proposed by Tresca et al. (2022) as T3CD. Specifically, we run experiments on the 140 instances we have created using the generator provided by Gzara et al. (2020). We compare these results with those reported by GEY and T3CD on their respective sets of generated instances, which have the same properties as ours but are not identical. To be able to compare with Gzara et al. (2020), we account for two additional features that they considered: different types of support surfaces and a limit on the maximum number of items that are considered at each iteration. The surface type indicates the shape of the item's top surface, as reported in Gzara et al. (2020). While boxes remain shaped as cuboids when evaluating the geometric constraints of the DPLP, a different surface type modifies the evaluation of static stability and load-bearing constraints. We include this feature in TBS by considering the different item shapes when computing the two-dimensional overlaps between each item and its supporting items. The limit on the maximum number of items to be considered per iteration is η. Specifically, the items are sorted according to a parameter called "planogram number", which is provided by the instance generator. At each iteration, the η available items with the smallest planogram number are considered for packing. In Gzara et al. (2020), this feature is implemented by imposing that no more than the first η items can be used to generate new layers at each iteration of their layer-based algorithm. In TBS, we mimic this feature by considering only the first η remaining items in the planogram sequence when branching in the beam search tree. As in Gzara et al. (2020), we consider a value of $\eta = 400$.

Table 4 provides a summary of the results which include the average number of bins, the average pack density, and the average computational time for each class of instances for the

			Bins (#)			Pack Density (%)			Time (s)		
$\|I\|$	Class	Groups (#)	GEY	T3CD	TBS	GEY	T3CD	TBS	GEY	T3CD	TBS
100	1	46.2	1.80	1.40	**1.00**	0.48	**0.80**	0.69	2.58	25.81	**0.85**
100	2	45.4	2.00	2.00	**1.00**	0.48	**0.75**	0.71	1.40	24.77	**0.49**
100	3	45.2	2.00	2.30	**1.00**	0.43	**0.76**	0.68	4.72	36.12	**0.56**
100	4	47.2	1.80	1.80	**1.00**	0.40	**0.79**	0.67	3.39	26.41	**0.50**
150	1	67.0	**2.00**	**2.00**	**2.00**	0.47	0.58	**0.68**	9.36	24.16	**0.78**
150	2	71.6	**2.00**	2.30	**2.00**	0.47	**0.76**	0.66	9.83	37.82	**0.96**
150	3	71.4	2.00	2.30	**1.20**	0.52	0.63	**0.67**	4.48	72.27	**1.00**
150	4	72.0	2.00	2.50	**1.40**	0.48	**0.64**	0.62	7.29	42.44	**1.31**
200	1	91.2	3.00	3.30	**2.00**	0.50	0.53	**0.72**	17.52	4.13	**1.19**
200	2	90.0	2.80	2.50	**2.00**	0.50	0.68	**0.73**	20.25	34.33	**1.31**
200	3	91.6	2.60	2.80	**2.00**	0.48	0.61	**0.66**	16.08	74.63	**1.69**
200	4	91.8	**2.00**	2.70	**2.00**	0.47	**0.75**	0.64	20.51	51.96	**1.54**
500	1	235.2	5.40	5.20	**4.00**	0.60	0.48	**0.78**	233.61	240.55	**5.26**
500	2	234.2	5.20	6.50	**4.00**	0.60	0.58	**0.78**	168.13	122.55	**5.90**
500	3	228.4	4.20	6.10	**3.80**	0.60	0.66	**0.68**	160.76	243.27	**5.43**
500	4	227.4	4.40	4.60	**3.00**	0.56	0.57	**0.77**	193.64	134.75	**6.29**
1000	1	471.4	9.40	9.00	**8.00**	0.66	0.69	**0.80**	373.32	470.85	**15.57**
1000	2	469.2	9.40	10.30	**8.00**	0.66	0.57	**0.79**	462.87	328.97	**19.71**
1000	3	463.6	8.00	8.40	**6.20**	0.62	0.54	**0.79**	646.68	250.13	**18.10**
1000	4	452.0	8.00	9.10	**6.00**	0.58	0.62	**0.79**	800.19	680.28	**20.14**
1500	1	696.0	13.80	13.50	**11.00**	0.67	0.69	**0.83**	1052.59	1478.15	**31.80**
1500	2	693.2	13.40	14.00	**11.00**	0.68	0.70	**0.83**	925.14	1631.39	**33.82**
1500	3	688.0	11.20	11.60	**9.00**	0.66	0.52	**0.82**	1140.45	609.62	**36.22**
1500	4	705.0	10.60	10.50	**8.60**	0.66	0.70	**0.80**	888.15	1031.65	**38.97**
2000	1	911.2	18.60	18.70	**15.00**	0.67	0.70	**0.83**	1675.10	2056.04	**56.03**
2000	2	904.0	18.40	18.60	**15.00**	0.67	0.68	**0.82**	2229.32	2315.45	**56.58**
2000	3	915.2	15.80	15.70	**12.00**	0.62	0.53	**0.83**	2687.51	2231.31	**55.91**
2000	4	913.4	15.40	15.60	**11.00**	0.59	0.79	**0.83**	3220.56	2398.11	**63.15**
Average			7.04	7.33	**5.51**	0.56	0.65	**0.75**	606.26	595.64	**17.18**

Table 4: Results reported by GEY, T3CD and TBS on instances created with the generator of Gzara et al. (2020)

three different algorithms TBS (this paper), GEY (Gzara et al., 2020), and T3CD (Tresca et al., 2022). The "Groups (#)" column reports the average number of distinct groups of items per instance, each consisting of items with the same dimensions, weight, and maximum load. This metric indicates the heterogeneity of item types within each problem instance, with higher values suggesting a greater diversity of item attributes. Each row corresponds to a group of 5 instances with the same number of items and the same class. Bold entries in each row indicate the best average result obtained for every measure. Computational times are reported in seconds and scaled by the theoretical peak performance of the CPU used to run the experiments. In particular, we use the number of floating point operations per second (FLOPS) as a measure of the CPU's performance, and we scale the elapsed time reported by Gzara et al. (2020) and Tresca et al. (2022) by the ratio between the theoretical peak performance of their CPU and the theoretical peak performance of the CPU used to run our experiments. According to specifications provided by the processors' manufacturer (Intel Corporation Ltd, 2024), the theoretical peak performance of the CPU used to run our experiments is 179.2 GFLOPS, while the theoretical peak performance of the CPUs used by Gzara et al. (2020) and Tresca et al. (2022) are respectively 76.8 GFLOPS and 211.2 GFLOPS. This implies that the elapsed times detailed in Table 4 for GEY and T3CD are equal to the original times reported in Gzara et al. (2020) and Tresca et al. (2022) multiplied by a factor of 0.428 and 1.178, respectively.

We observe that, on average, TBS improves the number of open bins by 22% (i.e., $\frac{5.51-7.04}{7.04}$) and 25% and the average pack density by 34% and 15%, compared to GEY and T3CD, respectively. Our algorithm provides the best number of open bins for all 28 groups of instances, and the best average pack density for 21 out of the 28 groups. From a computational time perspective, on average, TBS needs only 2.8% and 2.9% of the scaled computational time used respectively by GEY and T3CD to solve each instance. These results highlight the effectiveness of TBS in solving the DPLP and in achieving significantly better results than approaches from the literature. Foremost, TBS obtained a lower number of bins in the large majority of cases at a fraction of the computational time required by GEY and T3CD, thus making TBS highly effective in practice and most importantly for use in the industry (as we show in Section 5.4). In no case were GEY and T3CD more computationally efficient or obtained a lower number of bins than TBS. Finally, we note that while T3CD obtained a better pack density in certain cases with a low number of items (200 and less), for instances with a larger number of items TBS obtained significantly better pack density than both, T3CD and GEY.

5.4. Industrial instances

We test TBS on 30 large instances provided by our industrial partner. These instances and their corresponding solutions are available online[2]. Compared to the random instances created using the instance generator of Gzara et al. (2020), the instances considered in this section are larger, with a number of items ranging from 2020 up to 2941. We perform two experiments: the first considers a restricted set of feasible orientations for each item $O_i = \{wdh, dwh\}\ \forall i \in I$, while the second allows all six possible orientations $O_i = O\ \forall i \in I$. Table 5 presents the results of these experiments. The first two columns report the number of items and the number of different groups of items per instance, respectively. The next three columns report the number of open bins, the average pack density, and the computational time obtained by TBS for each instance when considering a restricted set of feasible orientations. The last three columns report the same values obtained by TBS when considering all orientations as feasible.

We observe that the objective function values improve when a larger set of feasible orientations for each item is considered. Specifically, the average number of open bins improves by 8.6% (i.e., $\frac{19.77-21.63}{21.63}$), and the average pack density improves by 6.8%. This improvement was expected, as having more feasible orientations effectively enlarges the feasible region.

We also observe that TBS can efficiently solve these industrial instances, taking an average computational time of 24.58 seconds when considering all orientations, and an average computational time of 17.27 seconds when dealing with a restricted set of feasible orientations. This proves that TBS is a strong methodology for practical use in the industry. We note that the computational time for solving the industrial instances when considering only two orientations is smaller than the average computational time needed by TBS to solve the smaller generated instances with 2000 items from Gzara et al. (2020). This is because these industrial instances are less heterogeneous than the ones from Gzara et al. (2020), as indicated in the column

[2]https://github.com/Daddeee/tetris-beam-search-instances

		Restricted orientations			**All orientations**		
$\|I\|$	Groups (#)	Bins (#)	Pack Density (%)	Time (s)	Bins (#)	Pack Density (%)	Time (s)
2020	50	18	0.76	13.54	17	0.79	21.39
2033	61	19	0.76	9.19	16	0.83	15.71
2043	97	26	0.70	16.32	23	0.77	23.90
2052	78	21	0.76	12.43	21	0.74	25.77
2062	69	21	0.74	12.69	17	0.80	20.30
2067	83	22	0.71	18.34	20	0.76	20.91
2081	49	15	0.79	11.71	14	0.86	14.22
2089	76	16	0.72	17.02	16	0.75	23.62
2102	92	20	0.74	17.66	19	0.79	21.73
2111	75	22	0.74	16.75	19	0.79	18.94
2193	71	20	0.74	16.89	18	0.80	20.45
2232	112	24	0.78	17.00	23	0.78	28.11
2232	55	21	0.77	10.05	21	0.77	18.04
2236	61	18	0.74	13.13	17	0.82	21.20
2247	96	22	0.69	18.67	19	0.77	24.10
2288	65	23	0.70	15.51	21	0.77	25.37
2295	76	25	0.72	19.68	22	0.74	27.30
2327	69	20	0.78	14.30	17	0.86	18.65
2330	84	20	0.77	19.41	21	0.75	28.90
2363	78	27	0.71	21.86	22	0.81	24.80
2380	80	19	0.75	19.34	23	0.75	33.01
2395	55	19	0.71	19.80	18	0.78	29.43
2473	52	17	0.67	23.18	14	0.80	23.29
2493	79	24	0.75	18.30	21	0.79	25.98
2502	63	22	0.75	19.33	21	0.78	35.85
2581	115	27	0.73	22.05	24	0.78	39.64
2627	73	23	0.74	13.85	20	0.84	21.18
2749	83	32	0.70	23.88	26	0.80	23.96
2843	98	24	0.74	23.21	24	0.75	31.05
2941	78	22	0.78	22.88	19	0.82	30.62
Average		21.63	0.74	17.27	19.77	0.79	24.58

Table 5: Results on industrial instances.

"Groups (#)". In fact, the average number of items with identical characteristics is 199.52, while for the instances obtained with the Gzara et al. (2020) generator it is 2.17. Having less heterogeneous instances allows TBS to place many items in a single state expansion, which reduces the number of states to be explored and eventually leads to less computational time.

6. Conclusions

We develop the Tetris Beam Search (TBS), a novel algorithm designed to address the Distributor's Pallet Loading Problem (DPLP), particularly in scenarios involving highly heterogeneous items. Our approach integrates a new constructive heuristic, the Tetris Heuristic (TH), with a beam search framework to efficiently solve DPLPs by dynamically constructing compact and non-compact item layers within pallets. We introduce several algorithmic enhancements which significantly improve the computational efficiency of TBS, and perform extensive computational experiments to demonstrate its effectiveness. These experiments show that our algorithm outperforms existing competing methods not only in terms of solution quality, improving the average number of open bins by 22% and the average pack density by 15%, but also in terms of computational efficiency, remarkably using less than 4% of their computational time on average. Finally, we demonstrate that TBS can effectively handle complex real-world conditions, where rapid computational times are generally required. Future research could explore techniques to improve the evaluation of partial states in the Beam Search tree. Moreover, TBS could be adapted to other types of three-dimensional packing problems, such as container loading or truck loading, where similar challenges are encountered.

Acknowledgement

This research was supported by E80 Group S.p.a. Joe Naoum-Sawaya would like to acknowledge the financial support of the Department of Elettronica, Informazione e Bioingeneria of Politecnico di Milano that made this collaboration possible. Joe Naoum-Sawaya was also supported by NSERC Discovery Grant RGPIN-2024-04176. Finally, we would like to thank the authors of Gzara et al. (2020) for providing us with their instance generator that was used to create the instances for our benchmarks.

References

Ali, S., Ramos, A.G., Carravilla, M.A., Oliveira, J.F., 2022. On-line three-dimensional packing problems: A review of off-line and on-line solution approaches. *Computers & Industrial Engineering* 168, 108122.

Alonso, M.T., Alvarez-Valdes, R., Parreño, F., Tamarit, J.M., 2016. Algorithms for pallet building and truck loading in an interdepot transportation problem. *Mathematical Problems in Engineering* 2016, 1–11.

Ancora, G., Palli, G., Melchiorri, C., 2020. A hybrid genetic algorithm for pallet loading in real-world applications. *IFAC-PapersOnLine* 53, 10006–10010. 21st IFAC World Congress.

Araya, I., Riff, M.C., 2014. A beam search approach to the container loading problem. *Computers & Operations Research* 43, 100–107.

Birgin, E.G., Lobato, R.D., Morabito, R., 2010. An effective recursive partitioning approach for the packing of identical rectangles in a rectangle. *Journal of the Operational Research Society* 61, 306–320.

Bischoff, E., Ratcliff, M., 1995. Issues in the development of approaches to container loading. *Omega* 23, 377–390.

Bischoff, E.E., Marriott, M.D., 1990. A comparative evaluation of heuristics for container loading. *European Journal of Operational Research* 44, 267–276. Cutting and Packing.

Bortfeldt, A., 2012. A hybrid algorithm for the capacitated vehicle routing problem with three-dimensional loading constraints. *Computers & Operations Research* 39, 2248–2257.

Bortfeldt, A., Wäscher, G., 2013. Constraints in container loading – a state-of-the-art review. *European Journal of Operational Research* 229, 1–20.

Cacchiani, V., Iori, M., Locatelli, A., Martello, S., 2022. Knapsack problems — an overview of recent advances. part ii: Multiple, multidimensional, and quadratic knapsack problems. *Computers & Operations Research* 143, 105693.

Ceschia, S., Schaerf, A., Stützle, T., 2013. Local search techniques for a routing-packing problem. *Computers & Industrial Engineering* 66, 1138–1149.

Chevalier, S., 2022. Global retail e-commerce sales 2014-2026. URL: `https://www.statista.com/statistics/379046/worldwide-retail-e-commerce-sales/`.

Crainic, T.G., Perboli, G., Tadei, R., 2008. Extreme point-based heuristics for three-dimensional bin packing. *INFORMS Journal on Computing* 20, 368—384.

Dell'Amico, M., Magnani, M., 2021. Solving a real-life distributor's pallet loading problem. *Mathematical and Computational Applications* 26, 53.

Deplano, I., Lersteau, C., Nguyen, T.T., 2021. A mixed-integer linear model for the multiple heterogeneous knapsack problem with realistic container loading constraints and bins' priority. *International Transactions in Operational Research* 28, 3244–3275.

Elhedhli, S., Gzara, F., Yildiz, B., 2019. Three-dimensional bin packing and mixed-case palletization. *INFORMS Journal on Optimization* 1, 323–352.

Fanslau, T., Bortfeldt, A., 2010. A tree search algorithm for solving the container loading problem. *INFORMS Journal on Computing* 22, 222–235.

Gendreau, M., Iori, M., Laporte, G., Martello, S., 2006. A tabu search algorithm for a routing and container loading problem. *Transportation Science* 40, 342–350.

Gonçalves, J.F., Resende, M.G., 2012. A parallel multi-population biased random-key genetic algorithm for a container loading problem. *Computers & Operations Research* 39, 179–190.

Gunawardena, K., Wijayanayake, A., Kavirathna, C., 2021. Solve manufacturer's pallet loading problem (MPLP) with practical warehouse constraints, in: 2021 6th International Conference on Information Technology Research (ICITR), 1–6.

Gzara, F., Elhedhli, S., Yildiz, B.C., 2020. The pallet loading problem: Three-dimensional bin packing with practical constraints. *European Journal of Operational Research* 287, 1062–1074.

Hodgson, T.J., 1982. A combined approach to the pallet loading problem. *A I I E Transactions* 14, 175–182.

Intel Corporation Ltd, 2024. APP Metrics for Intel Microprocessors. Technical Report. Intel Corporation Ltd. URL: `https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Core-Processors.pdf`. last reviewed: 2024-01-18.

Iori, M., Loti de Lima, V., Martello, S., Monaci, M., 2022. 2DPackLib: a two-dimensional cutting and packing library. *Optimization Letters* 16, 471–480.

Junqueira, L., Morabito, R., Sato Yamashita, D., 2012. Three-dimensional container loading models with cargo stability and load bearing constraints. *Computers & Operations Research* 39, 74–85. Special Issue on Knapsack Problems and Applications.

Kır, S., Yazgan, H.R., 2019. A novel hierarchical approach for a heterogeneous 3D pallet loading problem subject to factual loading and delivery constraints. *European Journal of Industrial Engineering* 13, 627–650.

Kirmse, A., 2004. Game Programming Gems 4. Delmar Thomson Learning.

Lowerre, B.P., Reddy, B.R., 1976. Harpy, a connected speech recognition system. *The Journal of the Acoustical Society of America* 59, S97–S97.

Morabito, R., Morales, S., 1998. A simple and effective recursive procedure for the manufacturer's pallet loading problem. *Journal of the Operational Research Society* 49, 819–828.

do Nascimento, O.X., Alves de Queiroz, T., Junqueira, L., 2021. Practical constraints in the container loading problem: Comprehensive formulations and exact algorithm. *Computers & Operations Research* 128, 105186.

Neuenfeldt Júnior, A., Silva, E., Francescatto, M., Rosa, C.B., Siluk, J., 2022. The rectangular two-dimensional strip packing problem real-life practical constraints: A bibliometric overview. *Computers & Operations Research* 137, 105521.

Paquay, C., Limbourg, S., Schyns, M., Oliveira, J.F., 2018. MIP-based constructive heuristics for the three-dimensional bin packing problem with transportation constraints. *International Journal of Production Research* 56, 1581–1592.

Paquay, C., Schyns, M., Limbourg, S., 2014. A mixed integer programming formulation for the three-dimensional bin packing problem deriving from an air cargo application. *International Transactions in Operational Research* 23, 187–213.

Pažitnov, A.L., 1984. Tetris.

Ramos, A.G., Oliveira, J.F., Gonçalves, J.F., Lopes, M.P., 2016. A container loading algorithm with static mechanical equilibrium stability constraints. *Transportation Research Part B: Methodological* 91, 565–581.

Silva, E., Leão, A., Toledo, F., Wauters, T., 2020. A matheuristic framework for the three-dimensional single large object placement problem with practical constraints. *Computers & Operations Research* 124, 105058.

Silva, E., Oliveira, J.F., Wäscher, G., 2014. The pallet loading problem: a review of solution methods and computational experiments. *International Transactions in Operational Research* 23, 147–172.

Sørensen, T., Foged, S., Gravers, J.M., Janardhanan, M.N., Nielsen, P., Madsen, O., 2016. 3D Pallet Stacking with Rigorous Vertical Stability, in: Distributed Computing and Artificial Intelligence, 13th International Conference. Springer International Publishing, 535–543.

Steudel, H.J., 1979. Generating pallet loading patterns: A special case of the two-dimensional cutting stock problem. *Management Science* 25, 997–1004.

Tresca, G., Cavone, G., Carli, R., Cerviotti, A., Dotoli, M., 2022. Automating bin packing: A layer building matheuristics for cost effective logistics. *IEEE Transactions on Automation Science and Engineering* 19, 1599–1613.

Wang, L., Guo, S., Chen, S., Zhu, W., Lim, A., 2010. Two natural heuristics for 3D packing with practical loading constraints, in: PRICAI 2010: Trends in Artificial Intelligence. Springer Berlin Heidelberg, Berlin, Heidelberg. Lecture notes in computer science, 256–267.

Wäscher, G., Haußner, H., Schumann, H., 2007. An improved typology of cutting and packing problems. *European Journal of Operational Research* 183, 1109–1130.

Appendix A. Notation tables

Sets	
I	Set of items to be packed
B	Set of identical bin
O	Set of possible items orientations
$O_i \subseteq O$	Set possible items orientations for item $i \in I$

Parameters	
W, D, H	dimensions of each bin
M	maximum weight supported by each bin
w_i, d_i, h_i	dimensions of each item $i \in I$
μ_i	weight of each item $i \in I$
m_i	maximum weight supported by each item $i \in I$
β	tolerance on the vertical gap between supporting items
α	minimum percentage of each item's base area that must be supported for static stability

Variables	
x_i, y_i, z_i	coordinates of the bottom-left-front corner of each item $i \in I$
x'_i, y'_i, z'_i	coordinates of the rear-right-top corner of each item $i \in I$
u_b	binary variables indicating if bin $b \in B$ is used
v_{ib}	binary variables indicating if item $i \in I$ is placed in bin $b \in B$
$\rho^x_{ij}, \rho^y_{ij}, \rho^z_{ij}$	binary variables indicating if item $i \in I$ is placed respectively behind, on the right or above $j \in I$
r_{oi}	binary variables indicating if item $i \in I$ is placed with orientation $o \in O$
z^{max}_b	maximum height reached by items placed in bin $b \in B$
s^x_{ij}, s^y_{ij}	length of the overlap segment between items $i, j \in I$ along the x and y axis, respectively
z^c_{ij}	binary variable indicating if item i can support item j
g_i	binary variable indicating if item i is placed on the ground
t^b_{ij}	binary variable indicating if both i and j are placed in bin b
l_{ij}	load exerted by item i on item j
q_i	total load exerted by other items on i

Tetris Heuristic	
Γ	set of items packed in the bin by TH
P	set of horizontal planes in the bin
z_p	vertical coordinate of plane $p \in P$
L_p	set of items supporting plane $p \in P$
T_p	set of items intersecting plane $p \in P$
Γ_p	set of items packed by TH on plane $p \in P$ in a single iteration
E	set of candidate points on a plane $p \in P$
R	rectangles to be packed by the 2DOKP algorithm, obtained from all combinations of items to pack and their feasible orientations

Tetris Beam Search	
S	set of states maintained at each TBS iteration
s_{in}	initial state
F	set of final states
s_{out}	the best final state according to DPLP's objective function
U_s	set of children states obtained from s with the *ExpandState* procedure
B_s	set of open bins in state s
Γ_b^s	set of items packed in bin $b \in B_s$ for state $s \in S$
Γ_s	set of all items packed in state s
P_b^s	set of horizontal planes in bin $b \in B_s$ for state $s \in S$
P_s	set of all horizontal planes in state s
G	set containing all partitions of set R, where each partition contains items that have the same height

Appendix B. SmoothCD function

Algorithm 6 describes the *SmoothCD* (Kirmse, 2004) function that we use to smoothly transition from the pack density of a parent state s_p towards the pack density of a child state s. It takes the following parameters:

- $g(s_p)$: the pack density of the parent state;
- $g(s)$: the pack density of the child state;
- v: the update velocity, initially set to 0 and then updated with each parent-child transition;
- t: the smoothing time delay, set to 1.5;
- δ: the time increment, set to 1.

The function returns as output the smoothed pack density of the child state $g'(s)$, and the velocity v to be used in subsequent updates.

Algorithm 6: *SmoothCD*

```
  input  : $g(s_p), g(s), v, t, \delta$
  output: $g'(s), v$
1 $\omega \leftarrow \frac{2}{t}$
2 $x \leftarrow \omega\delta$
3 $exp \leftarrow \frac{1}{1+x+0.48x^2+0.235x^3}$
4 $\rho \leftarrow g(s_p) - g(s)$
5 $\tau \leftarrow (v + \omega\rho)\delta$
6 $v \leftarrow (v - \omega\tau)exp$
7 $g'(s) \leftarrow g(s) + (\rho + \tau)exp$
8 return $g'(s), v$
```

Appendix C. Computational results tables

				Time (s)		
N	Class	Bins (#)	Pack Density (%)	No BVH	No Matrix	TBS
100	1	1.00	0.69	0.61	0.96	0.85
100	2	1.00	0.71	0.46	0.68	0.49
100	3	1.00	0.68	0.50	0.89	0.56
100	4	1.00	0.67	1.21	1.15	0.50
150	1	2.00	0.68	1.84	2.86	0.78
150	2	2.00	0.66	1.86	2.85	0.96
150	3	1.20	0.67	3.03	3.28	1.00
150	4	1.40	0.62	2.70	3.82	1.31
200	1	2.00	0.72	2.83	3.93	1.19
200	2	2.00	0.73	3.88	4.35	1.31
200	3	2.00	0.66	4.68	6.35	1.69
200	4	2.00	0.64	3.67	7.22	1.54
500	1	4.00	0.78	14.89	17.06	5.26
500	2	4.00	0.78	15.80	18.27	5.90
500	3	3.80	0.68	14.51	20.15	5.43
500	4	3.00	0.77	17.63	21.37	6.29
1000	1	8.00	0.80	43.10	43.96	15.57
1000	2	8.00	0.79	57.97	46.36	19.71
1000	3	6.20	0.79	55.84	49.12	18.10
1000	4	6.00	0.79	63.80	51.24	20.14
1500	1	11.00	0.83	96.24	75.16	31.80
1500	2	11.00	0.83	104.80	82.22	33.82
1500	3	9.00	0.82	121.07	90.48	36.22
1500	4	8.60	0.80	134.94	96.52	38.97
2000	1	15.00	0.83	185.18	127.59	56.03
2000	2	15.00	0.82	193.95	132.51	56.58
2000	3	12.00	0.83	201.21	125.32	55.91
2000	4	11.00	0.83	228.46	134.13	63.15
Average		5.51	0.75	56.31	41.78	17.18

Table C.6: Results on instances from Gzara et al. (2020).

		Bins (#)						Pack Density (%)						Time (s)					
N	Class	$k=1$	$k=5$	$k=10$	$k=20$	$k=50$	$k=100$	$k=1$	$k=5$	$k=10$	$k=20$	$k=50$	$k=100$	$k=1$	$k=5$	$k=10$	$k=20$	$k=50$	$k=100$
100	1	1.2	1.0	1.0	1.0	1.0	1.00	0.58	0.69	0.69	0.72	0.75	0.77	0.14	0.31	0.85	0.69	1.45	3.55
100	2	1.2	1.0	1.0	1.0	1.0	1.00	0.58	0.68	0.71	0.72	0.77	0.79	0.09	0.17	0.49	0.46	1.63	3.86
100	3	1.0	1.0	1.0	1.0	1.0	1.00	0.61	0.65	0.68	0.69	0.74	0.72	0.08	0.18	0.56	0.69	1.92	3.77
100	4	1.0	1.0	1.0	1.0	1.0	1.00	0.61	0.68	0.67	0.70	0.74	0.73	0.06	0.18	0.50	1.37	2.18	4.60
150	1	2.0	2.0	2.0	2.0	2.0	2.00	0.60	0.66	0.68	0.68	0.73	0.71	0.10	0.31	0.78	2.20	3.50	9.55
150	2	2.0	2.0	2.0	2.0	2.0	2.00	0.62	0.63	0.66	0.66	0.65	0.66	0.10	0.33	0.96	1.96	5.10	12.39
150	3	2.0	1.4	1.2	1.0	1.0	1.00	0.52	0.61	0.67	0.75	0.76	0.79	0.12	0.67	1.00	2.23	4.01	11.45
150	4	2.0	1.4	1.4	1.2	1.0	1.00	0.45	0.63	0.62	0.66	0.75	0.76	0.12	0.71	1.31	2.46	4.11	9.83
200	1	2.0	2.0	2.0	2.0	2.0	2.00	0.65	0.71	0.72	0.75	0.74	0.77	0.16	0.83	1.19	2.31	6.59	11.75
200	2	2.0	2.0	2.0	2.0	2.0	2.00	0.68	0.70	0.73	0.76	0.74	0.76	0.13	0.98	1.31	2.96	7.17	11.99
200	3	2.0	2.0	2.0	2.0	2.0	2.00	0.63	0.68	0.66	0.68	0.68	0.69	0.17	0.86	1.69	2.76	8.79	16.67
200	4	2.0	2.0	2.0	2.0	2.0	2.00	0.63	0.65	0.64	0.66	0.67	0.65	0.31	0.85	1.54	4.54	9.19	15.39
500	1	5.0	4.2	4.0	4.0	4.0	4.00	0.65	0.75	0.78	0.80	0.81	0.81	1.14	3.34	5.26	11.07	30.75	43.34
500	2	5.0	4.0	4.0	4.0	4.0	4.00	0.65	0.77	0.78	0.81	0.80	0.81	1.04	3.47	5.90	11.37	30.51	51.88
500	3	4.0	3.8	3.8	3.4	3.6	3.00	0.69	0.72	0.68	0.75	0.71	0.82	0.93	4.13	5.43	11.37	28.41	48.63
500	4	4.0	3.2	3.0	3.0	3.0	3.00	0.62	0.73	0.77	0.77	0.79	0.80	0.93	3.29	6.29	12.04	26.42	48.96
1000	1	8.0	8.0	8.0	8.0	8.0	8.00	0.77	0.79	0.80	0.80	0.80	0.80	3.31	11.47	15.57	34.25	82.52	158.25
1000	2	8.0	8.0	8.0	8.0	8.0	8.00	0.77	0.79	0.79	0.79	0.79	0.78	3.26	11.08	19.71	40.27	77.59	154.28
1000	3	7.0	6.4	6.2	6.0	6.0	6.00	0.73	0.77	0.79	0.82	0.82	0.83	2.86	11.11	18.10	37.24	73.03	147.66
1000	4	6.0	6.0	6.0	6.0	6.0	6.00	0.75	0.79	0.79	0.80	0.81	0.82	2.94	9.96	20.14	36.47	80.94	152.13
1500	1	12.0	11.6	11.0	11.0	11.0	11.00	0.77	0.79	0.83	0.83	0.84	0.85	5.12	20.20	31.80	68.20	144.18	315.81
1500	2	12.0	11.2	11.0	11.0	11.0	11.00	0.77	0.81	0.83	0.83	0.84	0.84	5.31	20.39	33.82	70.04	135.31	302.14
1500	3	10.0	9.0	9.0	9.0	9.0	9.00	0.74	0.81	0.82	0.82	0.84	0.84	5.69	21.42	36.22	68.80	145.98	300.68
1500	4	9.0	8.8	8.6	8.2	8.0	8.00	0.77	0.79	0.80	0.82	0.84	0.84	6.40	23.84	38.97	79.91	160.87	343.97
2000	1	15.6	15.0	15.0	15.0	15.0	14.60	0.79	0.82	0.83	0.84	0.84	0.84	9.66	33.12	56.03	119.10	290.64	578.54
2000	2	15.2	15.0	15.0	14.4	14.2	14.20	0.79	0.81	0.82	0.84	0.85	0.85	9.66	34.93	56.58	118.00	261.15	520.04
2000	3	12.8	12.0	12.0	12.0	12.0	12.00	0.78	0.83	0.83	0.84	0.84	0.85	9.51	38.85	55.91	124.76	246.21	500.12
2000	4	12.0	11.0	11.0	11.0	11.0	11.00	0.77	0.83	0.83	0.84	0.84	0.85	10.43	39.90	63.15	119.70	298.74	601.98
Average		5.93	5.57	5.51	5.44	5.42	5.39	0.68	0.73	0.75	0.77	0.78	0.79	2.85	10.60	17.18	35.26	77.46	156.54

Table C.7: Results on instances from Gzara et al. (2020).

Instances			Bins (#)					Pack Density (%)					Time (s)				
N	Fam (#)	Dim (#)	$k = 1$	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 1$	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 1$	$k = 5$	$k = 10$	$k = 20$	$k = 50$
2043	69	97	26.00	27.00	26.00	23.00	24.00	0.71	0.69	0.70	0.77	0.75	1.37	6.26	15.23	27.67	105.68
2102	69	92	21.00	21.00	20.00	20.00	20.00	0.72	0.70	0.74	0.74	0.78	1.63	8.49	17.42	36.38	87.37
2232	69	112	29.00	26.00	24.00	25.00	24.00	0.68	0.74	0.79	0.76	0.79	2.02	6.39	15.15	35.55	94.01
2247	66	96	20.00	20.00	22.00	19.00	19.00	0.68	0.76	0.69	0.81	0.77	2.15	7.35	16.23	34.28	113.05
2493	49	79	26.00	25.00	24.00	23.00	23.00	0.68	0.71	0.75	0.74	0.74	4.05	8.73	15.17	36.75	118.16
2581	71	115	31.00	28.00	27.00	25.00	24.00	0.63	0.71	0.73	0.71	0.76	4.81	15.13	19.91	52.30	139.45
2749	53	83	37.00	33.00	31.00	31.00	29.00	0.65	0.69	0.71	0.74	0.76	3.16	8.73	23.40	41.35	133.45
2843	71	98	28.00	24.00	24.00	23.00	23.00	0.67	0.75	0.74	0.76	0.74	2.24	9.77	19.20	38.54	136.04
2941	55	78	22.00	24.00	22.00	22.00	22.00	0.69	0.71	0.78	0.77	0.78	1.94	12.84	19.77	52.99	157.27
3077	13	80	12.00	11.00	11.00	11.00	11.00	0.74	0.79	0.78	0.81	0.84	3.31	8.63	15.13	30.70	74.55
3117	19	46	11.00	10.00	10.00	10.00	10.00	0.75	0.79	0.81	0.82	0.82	2.08	9.26	18.61	27.22	91.98
3286	15	77	14.00	13.00	12.00	13.00	12.00	0.72	0.78	0.78	0.77	0.79	1.64	9.38	20.69	38.50	100.56
3488	14	83	12.00	12.00	12.00	11.00	12.00	0.77	0.80	0.81	0.85	0.82	3.36	12.33	16.05	38.37	109.57
3644	16	45	14.00	14.00	14.00	14.00	14.00	0.84	0.82	0.83	0.83	0.85	3.76	12.05	19.10	45.39	141.08
3901	16	88	15.00	14.00	14.00	13.00	13.00	0.71	0.74	0.79	0.79	0.80	5.81	15.62	23.81	49.32	141.53
3907	16	88	13.00	13.00	12.00	13.00	12.00	0.78	0.77	0.82	0.80	0.80	6.18	16.18	27.25	57.23	191.67
3968	21	45	15.00	15.00	12.00	13.00	13.00	0.73	0.76	0.80	0.77	0.81	3.65	16.15	34.72	65.92	242.84
3989	13	58	15.00	15.00	15.00	15.00	15.00	0.80	0.82	0.83	0.82	0.83	1.98	11.50	21.44	44.41	118.38
3993	14	59	15.00	15.00	15.00	15.00	15.00	0.83	0.84	0.86	0.84	0.87	2.88	12.56	25.18	46.57	120.96
4109	18	109	14.00	14.00	14.00	13.00	14.00	0.85	0.85	0.83	0.88	0.86	5.78	13.91	29.22	52.91	206.97
4123	18	109	14.00	14.00	13.00	14.00	13.00	0.86	0.84	0.86	0.84	0.86	4.60	11.46	26.59	60.31	243.93
4290	27	116	18.00	17.00	16.00	16.00	16.00	0.73	0.77	0.76	0.78	0.80	5.95	20.21	42.11	80.16	265.88
4484	18	110	15.00	15.00	14.00	15.00	14.00	0.82	0.79	0.83	0.83	0.86	3.67	18.91	34.67	77.44	269.03
4498	18	106	15.00	15.00	15.00	14.00	14.00	0.82	0.84	0.84	0.87	0.87	3.23	16.06	35.14	75.70	209.95
4531	25	122	18.00	18.00	17.00	17.00	17.00	0.75	0.77	0.77	0.80	0.78	6.03	21.72	38.51	71.76	251.63
4712	26	128	18.00	18.00	18.00	17.00	17.00	0.78	0.78	0.80	0.82	0.82	6.80	20.23	37.74	76.04	243.81
4734	26	135	19.00	18.00	18.00	19.00	18.00	0.78	0.78	0.79	0.77	0.80	5.41	20.26	35.78	72.91	243.96
4835	26	132	19.00	19.00	18.00	18.00	17.00	0.76	0.76	0.79	0.78	0.82	7.34	20.31	42.75	86.97	289.08
4920	28	141	19.00	19.00	18.00	18.00	18.00	0.75	0.75	0.80	0.79	0.79	6.77	22.16	40.27	92.17	285.92
5117	31	182	18.00	18.00	18.00	18.00	17.00	0.75	0.77	0.76	0.79	0.78	7.85	27.09	59.77	110.92	407.42
5139	29	142	20.00	19.00	19.00	19.00	18.00	0.74	0.76	0.78	0.79	0.80	5.29	26.11	49.15	124.50	330.66
5261	29	150	19.00	19.00	19.00	19.00	18.00	0.79	0.80	0.78	0.78	0.81	5.44	23.51	48.42	119.33	294.67
5296	29	152	19.00	20.00	19.00	19.00	18.00	0.72	0.73	0.76	0.77	0.79	7.99	27.57	49.54	107.10	321.47
5468	25	150	18.00	19.00	18.00	19.00	18.00	0.78	0.76	0.80	0.79	0.81	5.45	26.03	51.97	110.16	336.45
5659	27	155	20.00	19.00	19.00	20.00	20.00	0.75	0.74	0.75	0.75	0.76	7.49	29.50	67.35	145.15	496.19
5720	22	66	16.00	18.00	17.00	17.00	16.00	0.79	0.73	0.77	0.78	0.81	8.51	30.64	56.54	119.75	359.53
5734	23	106	16.00	17.00	17.00	16.00	17.00	0.81	0.80	0.78	0.80	0.79	6.14	25.08	50.67	107.67	321.36
5734	23	106	16.00	17.00	17.00	16.00	16.00	0.81	0.78	0.80	0.81	0.82	6.99	24.83	50.07	102.74	379.17
5847	28	75	19.00	19.00	19.00	19.00	18.00	0.76	0.77	0.77	0.77	0.79	4.95	24.90	50.31	106.31	341.08
6044	21	109	19.00	19.00	19.00	18.00	18.00	0.77	0.77	0.79	0.80	0.80	5.53	28.04	44.77	111.26	306.56
6061	24	132	19.00	19.00	19.00	19.00	18.00	0.76	0.76	0.74	0.77	0.78	8.13	27.55	58.19	108.07	344.99
6164	28	216	21.00	21.00	20.00	19.00	19.00	0.70	0.72	0.76	0.77	0.79	8.82	42.64	74.36	147.42	562.58
6243	23	98	20.00	20.00	20.00	19.00	19.00	0.72	0.74	0.73	0.74	0.75	6.23	30.79	80.36	147.02	462.55
6245	16	121	21.00	20.00	20.00	19.00	20.00	0.74	0.76	0.78	0.79	0.76	8.55	28.41	57.22	128.27	538.95
6410	26	144	21.00	21.00	20.00	20.00	20.00	0.73	0.74	0.74	0.76	0.76	9.79	35.45	64.90	125.50	413.62
6438	20	90	21.00	20.00	19.00	19.00	19.00	0.74	0.77	0.80	0.77	0.79	8.65	31.51	61.63	130.99	438.21
6472	22	140	19.00	19.00	20.00	19.00	19.00	0.74	0.75	0.76	0.78	0.75	9.44	32.21	58.53	137.00	382.32
6658	19	106	20.00	19.00	21.00	19.00	20.00	0.74	0.78	0.74	0.75	0.74	9.79	33.22	75.79	139.12	462.51
6739	23	104	20.00	18.00	19.00	17.00	17.00	0.69	0.73	0.73	0.78	0.77	11.82	45.98	80.28	185.10	647.32
Average			18.92	18.53	18.10	17.76	17.51	0.75	0.76	0.78	0.79	0.80	5.44	20.28	39.51	82.84	266.84

Table C.8: Results on industrial instances.