New FFD-based Algorithms for Multi-Dimensional Vector Packing: An Empirical Study

István Zoltán Kolozsi

University of Szeged, Faculty of Science, Institute of Informatics, Árpád tér 2, H-6720 Szeged, Hungary
kolozsi@inf.u-szeged.hu

János Balogh

University of Szeged, Institute of Informatics, Department of Computational Optimization, Árpád tér 2, H-6720 Szeged, Hungary
baloghj@inf.u-szeged.hu

Abstract: The vector-packing problem is essentially a multidimensional generalization of the well-known bin packing problem, which is a classical optimization problem, and we know that even the one-dimensional version is NP-hard. Due to this, several approximation algorithms and heuristics have been developed. The task of a company that offers Virtual Machine resources for rent to their users can be expressed as a d-dimensional Vector Bin-Packing problem. The Geometric Heuristics (Dot-Product and L2) algorithms have been shown in the literature to be effective, and they generally perform better than the FFD variants. Therefore, we have designed new algorithms that are based on FFD, which should be able to compete with GH algorithms. They work by changing the original packing order of the FFD based on some set of rules. This proved to be a challenging task, but our FFDBox, FFDGroups, and FFDBG algorithms, which operate in a non-deterministic way (i.e., using random steps as well), achieved promising results. We tested all the algorithms on various input (benchmark) examples in 1, 2, 3, 4, and 6 dimensions. The results obtained are given here.

Keywords: vector packing; cloud server computing; Geometric Heuristics; FFD-based algorithms; benchmark test examples

1 Introduction

For large companies that provide packaging services, the efficient use of resources is essential to remain competitive. For example, consider a company that offers Virtual Machine resources for rent to its users, where the allocation of tasks received by users is a fundamental and very important issue. This task can be formulated as a Vector Bin-Packing (VBP_d) problem.

In the paper, we will describe the importance, difficulties, and possible uses of the vector packing problem. We will analyze empirically the FFD-based algorithms presented in [15, 2] and the new ones we have developed. For the analysis, we also found and generated input data on which we performed the tests.

In the next section, we will give a brief summary of some of the algorithms available in the literature. After that, we will present some virtual machine features that can help provide an efficient solution to the vector packing problem. Then, we will introduce the VBP_d problem, the family of FFD algorithms, Geometric Heuristics, and FFD-based algorithms that we have designed.

Following a literature overview, we will outline the structure of the new algorithms designed by us and define benchmark examples we use. In the final part of the paper, we present computational results where we compare the algorithms in question using tables and analyse their average empirical behaviour.

1.1 Literature Overview

The paper of Panigrahy et al. [15] is the main inspiration and basis for our study. They presented examples that make use of Geometric Heuristics (Dot-Product and L2 algorithms) and Grasp [K]. They performed empirical analyses on tasks with different dimension numbers and compared the results of their algorithms with known FFD variants. The authors concluded that they had succeeded in creating heuristics that are useful in practice; e.g., they mention that Microsoft's Virtual Machine Manager [13] uses the Dot-Product and Norm-based Greedy heuristics. These, together with the above methods, will be discussed in more detail later.

In their study [1], Brandao and Pedroso present an efficient but computationally expensive solution to the VBP_d and cutting stock problem. They constructed a solution graph for the problem and iteratively ran the graph reduction method they had defined on this graph, which can be treated as an arc-flow problem. However, their method is only applicable to integer vectors and gives an exact solution, so it is not suitable for solving large-scale problems. The problem is NP-hard, as will be discussed later, and we do not yet know for certain whether a method can be applied that gives exact solutions.

The following study is worth mentioning because it explains how important the scheduling problem is for Virtual Machines [10]. The authors of the paper, Saikishor Jangiti and Shankar Sriram, emphasize that it is economically vital for companies to use their available resources as efficiently as possible.

The article entitled "The Three-Dimensional Bin Packing Problem" [12] contains three-dimensional examples that we will describe and use with some minor modifications. Since we deal with vector packing problems here, not all the input examples in this study are useful to us. The essential difference is that they allow dimension swapping (items can be rotated), but this is forbidden in the case of vector packing.

Nagel and his co-authors published a paper [14] in 2023, in which they investigated the vector scheduling and vector packing problems in great detail. They also stated as a conclusion that exact algorithms and approximation schemes are not suitable for practical purposes due to their long runtimes, which highlights the importance of the algorithms we have examined and defined. Several new algorithms have been defined and investigated, mostly focusing on the vector scheduling problem. Fewer algorithms have been investigated for the vector packing (VP) problem, which typically have a long runtime, like the CNS (consistent neighbourhood search) algorithm, and an LS (local search) algorithm that post-processes the output of other algorithms using a local search, which needs additional time, and the mixing of a bin- and item-centric approach. Their experiments and results lead to the overall conclusion that there is a general trade-off between runtime and packing quality.

Several of the above-mentioned studies were referred to and applied in the work of Caprara and Toth [2], where the authors defined several input classes for testing. We have drawn inspiration from them and reproduced the classes of the input examples they presented, together with other test examples. In their article, they focused on the 2-dimensional vector packing problem and followed two approaches. The first one is based on the computation of several lower bounds. Although the given lower bounds can be computed quickly, their application does not give efficient results. The other approach is to formalize an integer programming formula with a large number of variables for the problem. The column generation method is used to solve the relaxed linear programming problem, but it is CPU-intensive and can only be applied to small problems, as mentioned above, in the context of exact algorithms.

Improvements in the results of Geometric Heuristics [15] were also discussed in [5]. They ran iterative algorithms on the GH run results that attempt to reorder the already packed items and hence create a new, better packing arrangement. They worked on a different, multi-objective, 2-dimensional problem. We also drew inspiration from this, with a similar initial intention to investigate FFD-based vector packing heuristics.

1.2 Motivation for the Problem: A Brief Introduction to Virtual Machines (VMs)

Infrastructure as a Service (IaaS), a cloud-based service delivery model, provides on-demand access to remote computing resources available in various Cloud Data Centres (CDCs). Virtual Machines (VMs) are the basis for Cloud-based technologies that are replacing traditional computing infrastructures. Users rent VMs from well-known Cloud Service Providers (CSPs) to run their own applications, such as Amazon EC2, Google Compute Engine, and Microsoft Azure, to name but a few popular ones [10].

Amazon Elastic Compute Cloud (Amazon EC2) [18] is a flexible and scalable cloud computing service that allows users to create and manage virtual servers on Amazon Web Services. Microsoft Azure [19] is a cloud computing platform, and it offers a collection of more than 200 services. Azure allows developers and IT professionals to run tests and manage applications across the network of global data centres. The Google Compute Engine (GCE) [9] is a cloud-based infrastructure service that provides many VMs per user as needed, and it enables VMs to be created and ran in its data centres. These virtual machines may be Windows or Linux-based and have different computing resources such as CPU, RAM, and HD storage.

Energy-efficient management is an active area of research in CDCs that attempts to reduce operating costs. Physical machines (PMs) account for a significant share of CDC energy consumption, and even when idle, PMs continuously consume 60% of their total load [10]. Reducing energy consumption helps to minimize carbon emissions and their adverse environmental effects. Autonomous cloud managers try to assign VM requests to as few PMs as possible to minimize the operational cost, which can be viewed as a Vector Bin-Packing (VBP_d) problem. The d refers to the fact that it is d-dimensional, i.e., it can handle d different resources.

This is why the VM allocation problem, and therefore the VBP_d, is not only a mathematical and IT problem, but also an economic and environmental issue. In addition, scheduling vectors and packing vectors into bins is a fundamental problem in operations research [14]. The importance of this problem is indisputable, which is why the VBP_d problem is actively studied today.

2 VBP_d

In the case of the d-dimensional Vector Packing (or Vector Bin Packing) Problem (VBP_d), a set I consisting of n number of items $I^1, I^2, \dots, I^n$ is given, where each $I^l \in \mathbb{R}^d$. A valid packing is said to exist when I is partitioned into k sets (or bins) $B_1, \dots, B_k$, where for each $1 \leq j \leq k$ bin and each $1 \leq i \leq d$ dimension, the following condition needs to be satisfied:

$$\sum_{I^l : I^l \epsilon B_j} I_i^l \leq 1,$$

i.e., the sum of the items (vectors) in a bin cannot exceed the capacity of the bin – which is 1 in this definition – in any dimension.

The objective of the VBP$_d$ problem is to find a valid packing of all items while minimizing the number of bins used. For the one-dimensional case, VBP$_d$ is one such generalization of the Bin-Packing Problem (BPP), where the items and the bins are d-dimensional vectors.

We note that VBP$_d$ is NP-hard for all d values (in the strong sense) [8]. In the one-dimensional case, an asymptotic PTAS (polynomial-time approximation scheme) exists [17], but due to its excessively long running time, it is completely unusable in practice. However, when $d \geq 2$, the problem is APX-hard [15], and no asymptotic PTAS can be given, supposing that $P \neq NP$ [3]. If there were one (even for $d = 2$), it would mean that $P = NP$ [20].

This makes the task a good example of the Virtual Machine (VM) hosting problem, with multiple resource capacity constraints. In practice, this problem models, for example, static resource allocation, where a minimum number of servers with known capacity is needed to meet the demand for services.

3 The FFD Family of Algorithms

The various approaches to solving the VBP$_d$ problem described in the literature are usually members of the First Fit Decreasing (FFD) family of algorithms. They are generalizations of the classical one-dimensional FFD algorithm [4]. For the one-dimensional problem, the FFD algorithm first arranges the items in a non-increasing order based on their size, and then uses the First Fit (FF) algorithm, which places each item in the first (earliest created) bin that can fit the item, i.e., it does not exceed the maximum capacity of the bin even after it is loaded. If no such bin exists, it opens a new one and places the item into it. The FFD algorithm is very popular in practice [4]. Dósa et al. [6] have shown that FFD in the one-dimensional case finds a packing that uses at most $\mathrm{FFD(l)} \leq \frac{11}{9} \cdot \mathrm{OPT(I)} + \frac{6}{9}$ number of bins for each input I, where $\mathrm{FFD(l)}$ is the number of bins used by the FFD algorithm and $\mathrm{OPT(I)}$ is the number of bins of the optimal solution. In the case of $d = 1$, this approach proves to be very efficient, both formally and in practice. In [7], it was also found that for the multidimensional problem, the asymptotic competitive ratio of the FF algorithm (which can also be treated as an online algorithm, i.e., without ordering the items) depends on the number of dimensions, namely $d + 0.7$.

With the multidimensional vector packing problem, if we use an FFD-type algorithm, the FF algorithm for packing can be defined as above. However, the first phase, namely the ordering of the items, can be defined in several ways. This is implemented based on the weight function applied to the items in the heuristics. Based on the weight functions applied, a different algorithm variant can be identified. For instance, the following are commonly used [15, 2]:

$$w(I^l) = \prod_{i \le d} I_i^l \qquad \text{(FFDProd)},$$

$$w(I^l) = \sum_{i \le d} I_i^l \qquad \text{(FFDSum)},$$

$$w(I^l) = \frac{\sum_{i \le d} I_i^l}{d} \qquad \text{(FFDAvg)}.$$

We recall that here I^l is the l-th item of input I, where $1 \le l \le$ n. In the algorithms presented, we perform different operations based on the (one-dimensional) weight of the input items, which differs from the d-dimensional vector of the item, which we call the demand for the item (in the case of $d > 1$), and the components of the vector are called component-wise demands.

Note that the above formulas actually result in only two variants since for FFDSum and FFDAvg, the weights of the items always differ by a factor of d. However, a multiplier a_{i} could be used to calculate the sum of the members. For example, in [2] the authors mention that this may represent the importance or priority of each dimension (For simplicity, in our case we did not use these kinds of multipliers. We simply set all the a_i-s to 1, as the authors did in [2].)

Running time. The one-dimensional FFD algorithm can be easily implemented in $O(n^2)$ time, but another implementation of it can be constructed in $O(n \log n)$ [8]. In the multidimensional case, the running time is the sum of the two parts of the algorithm. First, there is the ordering of the items, which takes $O(nd + n\,log\,n)$ time. And to compute the time required to pack the items, we should recall that an algorithm that checks all the bins when it wants to pack an item has a running time of $\Omega(n \log n + nk)$, where k is the number of bins in the solution [15].

Bad instances to FFD. In the analysis of the algorithms (as mentioned above), if two or more items have the same weight, we assume that the item with the lower index is always packed first. This will play a key role in the bad instances discussed below, as they use items of the same weight but with different demand in the same dimension.

Example 1 [15]. Consider the two-dimensional example where we have $2n$ items, half of the items, i.e., n items are $(\frac{1}{3}, \frac{1}{6})$ and the other half are $(\frac{1}{6}, \frac{1}{3})$. In this case, the optimal solution loads 4 items per bin, two of each type, while any FFD variant loads 3 items per bin, and it produces a bin number of $\frac{4}{3}$ times the optimum.

Example 2. In the three-dimensional case, there are three different types of items whose resource requirements are, in order, $(\frac{1}{3},\frac{1}{3}-\varepsilon,\frac{1}{3}+\varepsilon)$, $(\frac{1}{3}-\varepsilon,\frac{1}{3}+\varepsilon,\frac{1}{3})$, and $(\frac{1}{3}+\varepsilon,\frac{1}{3},\frac{1}{3}-\varepsilon)$. Let each of the three types have $3n$ items, where n is an arbitrary positive even integer and ε is a very small fixed positive value. For this input, the optimal solution assigns one of each of these three types of items to a bin. However, the FFD variants mentioned above will not be able to do this because they will pack the same type into bins two at a time. This produces a packing where the number of bins is equal to $3/2$ times the optimum.

We can easily find instances that are worse than the above bad instances. In [15], a theorem states that the approximation ratio of FFD is at least $\left(1-\frac{1}{k}\right)\cdot d$ for any integer k. Their construction is based on the assumption that all the items Y of type T_i $(1 \leq \mathrm{i} \leq d)$ are identical, and have the common demand defined in the following manner:

$$Y_j = \begin{cases} \dfrac{1}{k}, & \text{if } j = i, \\ \dfrac{1}{(d-1)(k-1)k}, & \text{if } j \neq i. \end{cases}$$

FFD packs n items into n/k bins, while there exists a partitioning such that items can be packed into $\frac{n}{(k-1)d}$ bins.

We investigated this because it was one of the main motivations for our study. The bad instances above suggest that if many items have nearly equal weight, and distinguishable ones are received as input, they can spoil the efficiency of FFD-based algorithms.

4 Description of the Algorithms Studied

In the following, we will describe some algorithms mentioned in the literature [2, 15]. Among them, there are some non-FFD-based ones. We first give two versions of FFD that differ in their technical aspects, followed by the Geometric Heuristics described in [15], and finally our newly defined FFD-based algorithms. In the FFD versions, items are always sorted first in non-decreasing order based on their weight. We will also refer to this new order of items as the *ordered list of the items*. The algorithm variants always choose one item from this ordered list (which initially contains unpacked items), pack it by the FF rule, and maintain the ordered list of the remaining unpacked items.

4.1 The Item- and Bin-Centric Approach to FFD

According to the Item-Centric approach to FFD implementation, we take the first item from the ordered list and place it into the first bin that the item fits in, then pack all the remaining items one by one. If the item does not fit into any bins, then open a new empty bin and place the item. We repeat this procedure for the remaining items until all the items in the ordered list are packed. In contrast, with the Bin-Centric approach, FFD has only one open bin at a time, and for each step, it inserts the first unpacked item (with the largest weight) that can fit into the current open bin. If there is no such item, then the bin is closed, and the algorithm opens a new empty bin and repeats the above process for the *unpacked* items.

In multidimensional cases, determining the non-increasing order of the items and determining how to select the next item to be packed led us to modify the FFD strategy. This line of thinking can lead to modifications of FFD that may be competitive with the best and most commonly used methods in the literature [15]. In the following, we will first describe the main essential principles of these algorithms and then outline the heuristic strategies we designed by modifying the FFD.

4.2 Geometric Heuristics

4.2.1 Dot-Product (DotP for short)

This heuristic of [15] defines the "largest" item (i.e., the next item selected for packing) by maximizing the so-called pointwise multiplication between the vector of free capacities and the vector of demands of the items. Formally, at a time t, let $r(t)$ denote the remaining capacity vector of a currently open bin, obtained by subtracting the sum of the demand of the items already placed from the capacity of the bins. We place the item I^l into the bin that maximizes the pointwise multiplication $\sum_i I_i^l r(t)_i$ using the vector of remaining capacities without violating the capacity constraint. (That is, when doing so, we only consider possible bins for packing that can fit the item.) Here, we will always choose an item-bin pair for packing that is the best-fitting pair of possibilities and iterate this process until we have packed all the items.

4.2.2 Norm-based Greedy (L2)

This heuristic takes the difference between the vector I^l and the residual capacities $r(t)$ according to a certain norm, instead of the dot product. This algorithm, called L2 in the literature [15], always places an unpacked item I^l in such a way as to minimize the quantity $\sum_i \left(I_i^l - r(t)_i\right)^2$, and the assignment here of course must not violate the constraint on capacity. We note that, similar to the L2 algorithm, the use of the norms $|\cdot|_1$ and $|\cdot|_\infty$ explains why the algorithms are called L1 and LInf.

Studies [14,15] revealed that they did not perform better than L2, as shown using the examples they studied.

4.2.3 Grasp[k]

An extension of the above geometric heuristics with Grasp[k] is also described in [15]. Grasp[k] is a very simple concept that addresses the question of whether it is always the best choice for the algorithm to choose the step that seems optimal at the time (placing the item that is considered best based on the weight calculations). This method does not choose the best possible solution, but the k-th best solution. Choosing k as 1 is the same as running the basic algorithm. However, if the value of k is greater than 1, then the above is true; but our experience has shown that if the value of k is too large, then instead of a desired improvement, there is actually a small deterioration.

4.3 New FFD-based Algorithms

So far, we have described the classical FFD algorithms and the Geometric Heuristics (henceforth GH), which were discussed in [15]. However, we wanted to create FFD algorithms that slightly modify or "tweak" the packing order and have a similar running time to the FFD but with a favourable efficiency, like the GH.

The reasons for our choice were that we thought GHs would give better empirical results. As they choose a smarter packing order by always recalculating the weights that are responsible for the packing order of the items (not sticking to a predefined packing order). Empirical results in [15] tell us that GH-type algorithms behave better than the previous FFD-based ones on the input examples they are given and tested on. Our goal was to provide heuristics that are competitive in practice with the best GH-type heuristics reported in the literature, but at the same time, they are still generated by modifying FFD-based heuristics.

Method. We defined and tested a kind of FFD-like algorithm that modifies the predefined packing order of the items with a certain frequency by choosing from the end (or from any of the remaining unpacked items) and not necessarily selecting the next item to be packed based on the non-increasing order. The basic idea was to avoid the difficulty inherent in bad instances by sometimes breaking the predefined packing order, i.e., the order imposed by some weight function in the preparatory step of an FFD variant. Hence, the algorithms we defined are similar to FFD algorithms in that:

- At the beginning, in the preparatory step, the input is sorted using a weight function,
- For packing, the First Fit (FF) rule is applied.

However, the order of packed items will be changed according to a specific set of rules.

In the following, we define new algorithms designed using these constraints. Each algorithm operates under its own set of rules, and it has different parameters, which we will describe later. The effectiveness of the algorithms (and parameter settings) will be demonstrated by experimenting with benchmark examples.

As we learn through empirical analyses, the newly defined algorithms include some that are competitive with DP and L2 algorithms, and they are already used in practice (see [15] in the context of cloud servers).

The algorithms listed below are the resulting variants, whose names (FFDRatio, FFDVal, FFDGroups, FFDBox, FFDBG) tell us something about of the nature of the algorithms. We will first outline the general idea and background of the design of these algorithms. Then, we describe their exact operation and the parameters used to implement them.

4.3.1 FFDRatio

In the FFDRatio algorithm, we (or a user) can specify an arbitrary integer X. Using this value, it will have a $1/\mathrm{X}$ probability of packing the last unpacked item in the ordered list. This algorithm will no longer be deterministic, so we cannot specify exactly the packing order in advance. Based on our empirical experiments, we concluded that $\mathrm{X} = 15$ was an appropriate, good choice (i.e., $1/15$ probability of choosing small items) in our test examples, so we used this value during the test phase.

4.3.2 FFDVal

With this algorithm, we have a 50% chance of packing the unpacked items from the front of the ordered list, and a 25-25% chance of packing from the middle or the end of the ordered list (of unpacked items). Naturally, this algorithm has a higher chance of improvement if we have a very large number of items and a relevant difference between the items with large weight at the beginning of the ordered list. The median item in the middle of the ordered list ($\lceil n^*/2 \rceil$-th item in the order of the unpacked items, if we still have n^* unpacked items) and the smallest unpacked items with the smallest weight at the end of the ordered list (of the unpacked items).

4.3.3 FFDGroups

The basic idea of the FFDGroups algorithm is that we try to group (in other words, cluster) the items according to their weight. After sorting the items in a non-increasing order based on the sum of the items by their components (component-wise demands), we take a threshold number to form the groups and create a group of roughly equal size with a corresponding number of items. Group number 1 will contain the items at the beginning of the ordered list, while the last group (threshold number equal to the group number) will contain the items at the end of the ordered list. We take the groups formed one at a time (group 1 being the first) and pack all the items in that group in random order according to the First Fit (FF) rule. Once we have packed all the items in the current group, we move on to the next group and pack the items in that group randomly. The algorithm stops running when the last item in the last group has been placed into a bin. (To be precise, we rounded it down to avoid division problems. If K is the number of groups to be created, then our first $K-1$ groups contain $\lfloor n/K \rfloor$ items in a group, while the K-th group contains all the remaining elements from the input n items.)

```
Algorithm FFDGroups

Organise the items in descending order according to their weight. Calculate how many groups to
create (NumberOfGroups) and how many items are in each group (NumberOfItemsPerGroup).

for j=1 to NumberOfGroups do
        if j< NumberOfGroups (the last group is not reached)
                for i=1 to NumberOfItemsPerGroup do
                        Choose an item randomly from the group, pack it by FF rule, and
                        delete it from the group

        else (j= NumberOfGroups, i.e. we reached the last group)
                for i=1 to Number of remaining (unpacked) items do
                        Choose an item randomly from the group, pack it by FF rule, and
                        delete it from the group
```

Pseudocode 1 – Pseudocode of FFDGroups

4.3.4 FFDBox

Pseudocode 2 – Pseudocode of FFDBox

```
Algorithm FFDBox

Let n be the number of input items, and let BoxSize be the size of the box.

Organise the items in descending order according to their weight. Initialize the box with the first
BoxSize number of items of the ordered list

for i=BoxSize+1 to n do
        Choose an item randomly from the box, pack it by FF rule, and delete it from the box
        Move the i-th item of the ordered list into the box (e.g. in the place of the packed item)

for i=1 to BoxSize do (The unpacked items are now all in the box.)
        Choose an item randomly from the box, pack it by FF rule, and delete it from the box
```

The algorithm starts, just as in FFD, by sorting the items in a non-increasing order, obtaining our initial ordered list of the (unpacked) items. The next part can be thought of as having a box, into which we temporarily place a few items (the box can hold as many items as the user specifies in a parameter. In Section 5.6, we will provide some parameters that work well.

First, we fill our box with the quantity of items from the beginning of the ordered list (these items are removed from the ordered list of the remaining items, which now can be considered a queue). Then, we randomly select an item from the box and pack it into a bin by using the FF rule. This creates an empty space in the box, and this empty space is filled with the next item in the queue. We continue this until we run out of items (and near the end of the algorithm run, when there are no items left in the packing queue, we still have to pack the items in the box).

4.3.5 FFDBG

This algorithm was made by combining the FFDBox and FFDGroups algorithms. First, all the items are divided into groups based on FFDGroups, i.e., we cluster all of them according to their weight. However, within the groups, we do not pack the items in a completely random way but apply the boxed approach of FFDBox.

The parameters are chosen empirically. The well-chosen parameters can help us achieve better results. The choice of setting for all the algorithms with their tested options is described later in Section 5.6.

Now we will elaborate on the ideas and details we considered when implementing the algorithms. During the development and implementation of the algorithms, we used the PyCharm [16] development environment, and the algorithms were written in Python. All the completed code, result tables, and documents can be viewed on GitHub at https://github.com/Pityundra/Pakolas_tdk [11].

5 Benchmark Examples

To be able to verify the correctness and capabilities of the implemented algorithms, it is necessary to use appropriate test data sets. In the literature, the dataset structures used in the paper by Caprara and Toth [2] are often taken as a standard, e.g., in [15]. We constructed and used three types of input test example sets. Below, we will describe them.

5.1 Instances with 8 Classes

When designing these classes of test cases, we essentially followed the guidelines suggested in the literature. The parameters of the first 5 classes were implemented one by one, as described in [15]. Classes 6, 7, and 8 were taken from [12].

Table 1 shows the capacity of the bins and the minimum and maximum (component-wise) demand of the items in the input examples of these 8 classes. The items were generated by a random number generator with a uniform distribution from a closed interval between the minimum and maximum demand stated in the table. The above classes were created for 1, 2, 3, 4, and 6-dimensional examples, and the constraints in the table are for the one-dimensional case. If the dimension number is greater than one, we use the same constraints for each dimension (component-wise). Examples of 400, 800, and 1000 items were produced for each class. For each dimension, there were three examples generated from each class with three different item numbers (400-800-1000), resulting in 72 run results per dimension.

Table 1
Description of the first eight classes of input examples

Class	Bin capacity per dim.	Min. demand of an item	Max. demand of an item
class1	1000	100	400
class2	1000	1	1000
class3	1000	200	800
class4	1000	50	200
class5	1000	25	100
class6	100	1	50
class7	10	1	10
class8	40	1	35

5.2 Correlated Input Examples

For dimensions 2, 4, and 6, we generated input examples where the demands of the items by dimensions are correlated with each other. The input sizes were 400, 800, and 1000 items. We called these class9 and class10, and both classes have bins with a capacity of 150 in each dimension. In the 2-dimensional case, in class9, as the first-dimension demand of the item, we generated a random number $\mathrm{d1}$ from the closed interval $[20, 100]$. And for the second-dimension demand of the item, we used the value $\mathrm{d1}$ generated by defining the interval as $[\mathrm{d1} - 10, \mathrm{d1} + 10]$. This gave a positive correlation between the item dimensions (components) in class9.

For class10, we created a negative correlation between pairs of dimensions, as described below. In the 2-dimensional case, we generated the item's first dimensional demand d1 from the closed interval [20, 100], but afterwards we generated its second dimensional demand from the closed interval $[110 - \mathrm{d1}, 130 - \mathrm{d1}]$. In the 4 and 6-dimensional cases, we simply repeated the step described for the 2-dimensional case, with pair of the third and fourth dimensions and fifth and sixth dimensions. (These are carried out in the same way as the correlated data set was generated in [15].)

5.3 Input Examples with Known Optimum Values

In the table shown here (Table 2), we list the input classes where we know the value of the optimal solution in advance. We did not find any experiments on this kind of input in the literature, but for these examples, we know the precise distance from the optimum, and if the new algorithms can improve on the algorithms we have used so far, we will be able to better judge the effectiveness of the modifications. This type of input is not listed in classes.

For example, in Table 2, the first row means that we generate 10 test examples, where the bins have a capacity of 100 in each dimension, and all the items can be optimally packed in 10 bins. Here, the quantity of items is not known in advance, and their requirements are between 1 and the maximum capacity of the bins.

Table 2
Input examples with a known optimum value

No. of test cases	Optimum value	Max. cap. of bins
10	10	100
5	20	100
5	100	100
5	500	1000
5	1000	10000

5.5 Calculation of the Lower Bound

We also calculated a simple lower bound (LB) value for the test examples. The LB is obtained simply by summing the demand of items needed in each dimension and dividing it by the capacity of the bins; then we round this value up (component-wise) to get the minimum number of bins needed in that dimension to pack the items (and taking their maximum value among the dimensions in a multi-dimensional example [5]). It should be mentioned that this LB is not equal to the optimum in most cases, and it may even be far from it, because the demand of the items may leave gaps (free spaces) in the bins with optimal packing. However, if one of the algorithms attains the lower bound, we can say that an optimal solution has been found.

5.6 Description of Parameters used in Tests

When running the benchmark examples, we tested several parameters for the algorithms, as different parameter values may prove more effective on different types of input data. When presenting the results, we always selected the parameters that gave the best results for the given input example, and we arranged the results in tabular form.

Parameter settings used to test the algorithms (for instances with different dimension numbers):

GH algorithms (DotP and L2) – Grasp[k] parameter value: 2,3 and 4

FFDRatio – probability of packing the last unpacked item from the ordered list: 1/10 and 1/15

FFDGroups – NumberOfGroups (how many groups we divide the items into): 4, 6, 10, and 20

FFDBox – BoxSize (how many items are in the box): 3, 4, 5, and 6

FFDBG – BoxSize and NumberOfGroups: 4-4, 6-4, and 5-3

6 Analysis of the Computational Results

Our main analytical focus here is to compare the results of the algorithms given in the literature [15] with the results of the new FFD-based algorithms. In the tables below, we provide the computer test results obtained for the various test sets.

6.1 Results for the First Eight Benchmark Classes

In Table 3, we show a summary of the results of the first eight classes. We separate the results into 5 different cases. For the first case, our new algorithms can achieve similar results as the FFD and the GH (106 cases); unfortunately, in this case, we cannot make progress in the results. (In progress, we mean that one of the new algorithms we defined managed to pack the items into fewer bins than any of the other algorithms tested.) In 33 cases, we can achieve the optimum value (the algorithms give back the same result as the calculated lower bound). The 5th row is where one of the new algorithms gives the best result. The following row shows where the new algorithms give better results than the FFD, although the best results are provided by the GH. Finally, there are only 24 cases where the new algorithms cannot show any progress (less than 6,7% of the cases). Overall, we achieved some kinds of positive results in 64,7% of the cases (progress or achieving the optimum).

Table 3
Aggregate test results for the first eight classes

Total number of test cases	72	72	72	72	72	360
Number of dimensions	1D	2D	3D	4D	6D	Total
New algorithm achieved the best results of the FFD and GH algorithms	43	34	19	8	2	106
New algorithm attained the optimum value	25	8	-	-	-	33
New algorithm improves both FFD and GH	4	9	21	29	33	96

New algorithm improves FFD, but not GH	-	13	27	30	34	104
Could not be attained or improved	-	8	5	8	3	24

Tables 4, 5, and 6 contain details of the results. In the headers of the table, we can see the dimension number, the number of items in the examples, and which class the example belongs to. In the line of the Lower Bound, we can see the value of the LB (given in section 5.5). In the next three lines, there are the results of the FFD, DotP, and L2 algorithms (the algorithms were run with different parameters, and the best results were selected). Followed by the five new non-deterministic algorithms (FFDBox, FFDGroups, FFDBG, FFDRatio, FFDVal). We note that, since the algorithms we defined involve a random step, they may produce different results when we run them repeatedly, so we run the non-deterministic algorithms 100 times. For this reason, the tables show the average of the algorithm runs and, next to it, the best and worst run results obtained.

In Tables 4 and 5, we show 3 data sets based on the same set of rules. It is important to note that even though the examples were generated based on the same rule set, they are not comparable with each other, as the examples contain different items.

Overall, in the one-dimensional case, the previous algorithms of the literature perform very well, attaining the optimum or very close to the lower bound in many cases, making it difficult to improve on them, but we succeeded in 4 out of 72 test cases. Table 4 presents the runs on the class4 type data with 800 items per test, with 3 different input examples. In example_1, the FFD, DotP, and L2 algorithms reach the lower bound (optimum), as well as the FFDBox, FFDGroups, and FFDBG. In example_2, only the FFDGroups and FFDBG can achieve the lower bound value. In example_3, none of the algorithms achieved the lower bound; all of them scoring above 103.

Table 4
1-dimensional results for class4 input for 800 items, with 3 different input examples

	1D					
	class4_800					
	example_1		example_2		example_3	
Lower Bound	102		102		102	
FFD	102		103		103	
DotP,DotP-gp	102, 103		103, 103		103, 103	
L2, L2-gp	102, 103		103, 103		103, 103	
FFDBox	102.42	102-103	103.0	103	103.0	103
FFDGroups	102.82	102-103	102.92	102-103	103.12	103-104
FFDBG	102.36	102-103	102.99	102-103	103.0	103
FFDRatio	103.41	103-104	103.6	103-104	104.31	104-105
FFDVal	104.81	104-105	104.74	104-105	105.49	105-106

Table 5
2-dimensional results for 800 items with class8 input, with 3 different input examples

	2D					
	class8_800					
	example_1		example_2		example_3	
Lower Bound	368		363		368	
FFD	376		375		378	
DotP,DotP-gp	377, 379		374, 378		379, 381	
L2, L2-gp	376, 378		375, 379		379, 382	
FFDBox	375.41	375-376	375.0	375	378.19	377-379
FFDGroups	376.88	376-379	376.19	375-377	379.38	378-380
FFDBG	375.46	375-376	375.0	375	378.2	377-379
FFDRatio	379.44	378-381	379.14	377-381	382.6	382-384
FFDVal	397.61	396-405	406.68	398-418	409.43	405-419

In two-dimensional cases (see Table 5), it was much more difficult to reach the optimum, and the new algorithms produced more mixed results. However, it can be said that the new FFD-based algorithms performed very well. They are not likely to produce worse results than the known algorithms (and even if this is the case, the "gap" is minimal). We should also remark that in cases where we get the results obtained by other algorithms and fail to improve on them, the values may be even optimal that we do not know about. For example, in the three-, four-, and six-dimensional cases, the lower bound is not achieved even once. However, this could be due to the fact that the value of the lower bound may actually be well below the value of the optimum.

For the three-dimensional input examples, our test results indicate that in 67% of the cases we improved the results of the FFD variants reported in the literature, and in 26% of the cases we surpassed the results of the GH algorithms. In Table 6, three-dimensional examples can be seen with 3 different classes. At the class6 example with 1000 items, it is evident that the new algorithms give better results than the FFD, but the DotP provides the best result.

Based on our experience with the four-dimensional case, the new algorithms have a 40% probability of achieving the best overall performance and an 82% probability of outperforming the FFD approach. In the six-dimensional case, these probabilities increase to 46% and 93%, respectively.

Overall, the new algorithms are more likely to produce better results than the GH or FFD algorithms known so far for higher dimensions. Unfortunately, the results of FFDVal are generally not competitive, and the FFDRatio algorithm gives better results in some cases, but there is a large variation between the results obtained. We can say that FFDBox, FFDGroups, and FFDBG (which is a hybrid of the two) produce really good results.

Table 6
Three-dimensional results, 3 different input examples:
class6 (with 1000 items), class7 (with 400 items), and class8 (with 1000 items)

	3D					
	class6_1000		class7_400		class8_1000	
Lower Bound	259		220		452	
FFD	271		269		476	
DotP,DotP-gp	268, 269		269, 272		477, 482	
L2, L2-gp	270, 272		274, 272		476, 485	
FFDBox	272.34	270-274	269.42	268-271	475.98	475-477
FFDGroups	272.81	270-275	269.02	268-271	477.49	476-479
FFDBG	272.36	270-274	269.34	268-271	475.9	475-477
FFDRatio	276.83	275-280	270.86	269-273	484.78	482-488
FFDVal	288.93	286-290	295.24	293-305	501.05	498-506

We should also mention that if one runs the algorithms under similar conditions, due to their non-deterministic nature, one will not get exactly the same results, but presumably similar results in terms of averages.

6.2 Results on Correlated Input Examples

As we mentioned previously, class9 and class10 contain input examples where there is a correlation between the component-wise demands (dimensions) of an item. We generated our correlated (negative and positive) examples in 2, 4, and 6 dimensions, with data set sizes of 400, 800, and 1000 items. This results in 18 possible cases, and we generated 6 datasets for each case, leading to a total of 108 examples on which we evaluated our algorithms.

Table 7 presents a summary of the results on the correlated instances. In the case of positive correlation, we were able to improve the performance of FFD in 65% of the instances, and in 10 cases, one of the new algorithms achieved the best result. It is evident that as the dimensionality increases, it becomes increasingly challenging to match or surpass the performance of GH-type algorithms. For negatively correlated cases, our new algorithms performed substantially better overall, achieving the best results in 52% of the instances and never falling behind the GH algorithms. Notably, in the set of 18 six-dimensional examples, our new algorithms delivered the best performance in 16 cases, corresponding to nearly 90%.

Table 7
Summary of the results on the correlated instances

	class9 (positive correlation)			class10 (negative correlation)		
Number of dimensions	2D	4D	6D	2D	4D	6D
New algorithm achieved the best results of the FFD and GH algorithms	13	5	1	15	9	2
New algorithm improves both FFD and GH	5	3	2	3	9	16
New algorithm improves FFD, but not GH	-	10	15	-	-	-

6.3 Results for Inputs with Known Optimum Value

In the one-dimensional case, both FFD- and GH-based algorithms successfully overcame obstacles and have configurations that successfully attain the optimum for each test case. However, it can also be seen that our algorithms were competitive. For each of these examples, the optimal solution was found.

In the two-dimensional case, the GH-type algorithms come closest to the optimal result, but they do not do it very often. They very often use opt+1 bins (opt here denotes the optimal number of bins). For larger optimum cases (i.e., more items to pack), the FFD algorithms fall further behind the GH algorithms. The results of our new FFD-based algorithms lie between those of GH and FFD. In several cases, the results of the L2 algorithm produced the best results, but it could be argued that in every case, the L2 algorithm consistently produced better results than FFD.

In the three-dimensional examples, it is seen that there where 10 bins that are the optimum (about 40-50 items in one example), the new FFD-based algorithms usually improve the results not only for the FFD, but also for the GH-like algorithms and even hit the optimum multiple times.

Conclusion

We studied a problem with a key practical application nowadays (in the area of data centres, cloud technology, and cloud computing), called the multidimensional vector packing problem. We reviewed the algorithms found to be the best in the literature, and we also designed new algorithms that have the efficiency of FFD-based algorithms, while striving to overcome the difficulties of their bad instance. We tested the best algorithms from the literature alongside the algorithms we constructed. Overall, we succeeded in creating algorithms with promising results. For different numbers of dimensions and task sizes, they proved to be competitive in several cases compared to the best algorithms described in the literature.

Acknowledgement

The work of J. Balogh was partially supported by the grant TKP2021-NVA-09 of the Ministry for Innovation and Technology.

References

[1] F. Brandao, J. P. Pedroso, Bin packing and related problems: General arc-flow formulation with graph compression, Computers & Operations Research, Vol. 69, pp. 56-67, 2016

[2] A. Caprara, P. Toth, Lower bounds and algorithms for the 2-dimensional vector packing problem, Discrete Applied Mathematics Vol. 111, Issue 3, pp. 231-262, 2001

[3] S. Chawla, Lecture Notes (Approximations Algorithms), URL: https://pages.cs.wisc.edu/~shuchi/courses/880-S07/scribe-notes/lecture05.pdf Last access: 20 May 2024

[4] E. G. Coffman, M. R. Garey, D. S. Johnson, Approximation algorithms for bin packing: a survey. In: Approximation algorithms for NP-hard problems, pp. 46-93, 1996

[5] N. Dahmani, F. Clautiaux, S. Krichen, El-Ghazali Talbi, Iterative approaches for solving a multi-objective 2-dimensional vector packing problem, Computers & Industrial Engineering, Vol. 66, Issue 1, 158-170, 2013

[6] Gy. Dósa, R. Li, X. Han, Zs. Tuza, Tight absolute bound for First Fit Decreasing bin-packing, Theoretical Computer Science, Vol. 510, pp. 13-61, 2013

[7] M. R. Garey, R. L. Graham, D. S. Johnson. Resource constrained scheduling as generalized bin packing. Journal of Combinatorial Theory Series A, 21, 3, 257-298, 1976

[8] M. T. Goodrich, Bin Packing, CS 165, Lecture Notes, URL: https://ics.uci.edu/~goodrich/teach/cs165/notes/BinPacking.pdf Last access: 20 May 2024

[9] Google Compute Engine, URL: https://www.techtarget.com/searchaws/definition/Google-Compute-Engine Last access: 20 May 2024

[10] S. Jangiti, S. Sriram, Scalable and direct vector bin-packing heuristic based on residual resource ratios for virtual machine placement in cloud data centers, Computers & Electrical Engineering, Vol. 68, pp. 44-61, 2018

[11] I. Z. Kolozsi, GitHub repository, URL: https://github.com/Pityundra/Pakolas_tdk Last access: 20 May 2024

[12] S. Martello, D. Pisinger, D. Vigo, The Three-Dimensional Bin Packing Problem, Operations Research, Vol. 48, No. 2, pp. 256-267, 2000

[13] Microsoft Systems Center Virtual Machine Manager, URL: http://www.microsoft.com/systemcenter/virtualmachinemanager. Last access: 20 May 2024

[14] L. Nagel, N. Popov, T. Süß, Z. Wang, Analysis of Heuristics for Vector Scheduling and Vector Bin Packing. In: Learning and Intelligent Optimization: 17th International Conference, LION 17, Nice, France, June 4-8, 2023, pp. 583-598

[15] R. Panigrahy, K. Talwar, L. Uyeda, U. Wieder, Heuristics for Vector Bin Packing for Microsoft Research Silicon Valley, Microsoft's VMM product group, 2011. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2011/01/VBPackingESA11.pdf Last access: 20 May 2024

[16] Pycharm webpage. URL https://www.jetbrains.com/pycharm/?var=1 Last accessed on 20 May 2024

[17] V. V. Vazirani. Approximation algorithms, Springer, 2003, ISBN 978-3-642-08469-0 ISBN 978-3-662-04565-7 (eBook) DOI: 10.1007/978-3-662-04565-7

[18] What is Amazon EC2? URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html Last access: 20 May 2024

[19] What is Azure? URL: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure Last access: 20 May 2024

[20] G. J. Woeginger, There is no asymptotic PTAS for two-dimensional vector packing, Information Processing Letters, 64, 6, 293-297, 1997