3-D Container Packing with Weight and Multi Drop constraints

A Project Report

submitted by

MANISHNANTHA M

in partial fulfilment of requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS

May 2023

THESIS CERTIFICATE

This is to certify that the thesis titled **3-D Container Packing with Weight and Multi Drop constraints** , submitted by **Manishnantha M**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. N. S. Narayanaswamy
Research Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date: May 19, 2023

ACKNOWLEDGEMENTS

I would like to thank my guide Prof. N. S. Narayanaswamy for his direction, support, and feedback throughout this project.

I would also like to thank my classmates Shashwathy Kannan (CS19B041) and K. R. Hariharan (CS19B079) for their insightful discussions that helped me refine my ideas.

Finally, I would like to express my gratitude to my parents for their constant and unconditional support.

ABSTRACT

KEYWORDS: Bin Packing, 3-D Packing, Logistics, Tree Search, Greedy

The bin packing problem is an optimization problem, in which we pack items of different sizes into a set of bins of finite capacity, with the objective of minimizing the number of bins used. The 3-D geometric bin packing problem is a specific case of the bin packing problem which has applications in logistics, specifically in packing objects into cargo containers. The problem is NP-Hard, hence there is no known algorithm that can solve it in polynomial time. Multiple techniques such as LP Formulation, Genetic Algorithms, etc. have been used over the years to tackle this problem. In this project, we propose and analyse a Tree Search heuristic to pack objects with weight and destination constraints into a container while maximizing the volume utilisation of the container.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

VU	Volume Utilisation
PIR	Packed to Input Ratio
SD	Standard Deviation
LP	Linear Programming
OR	Operations Research
1-DBP	One Dimensional Bin Packing
2-DBP	Two Dimensional Bin Packing
3-DBP	Three Dimensional Bin Packing

NOTATION

t_w	Tree Width
n	Number of items to be packed
d_m	Maximum dimension among all items
$\mathbf{T}_x$	Time complexity of an Algorithm x
μ_k	Mean of a variable k
σ_k	Standard Deviation of a variable k
$G[i, j]$	Element in the i^{th} row and j^{th} column of a 2-D grid G

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

The bin packing problem is an optimization problem, in which we pack items of different sizes into a set of bins of finite capacity, with the objective of minimizing the number of bins used. As mentioned in Khan (2015), in the classical bin packing problem, we are given a list of real numbers in the range $(0, 1]$, and the goal is to place them in a minimum number of bins so that no bin holds numbers summing to more than 1.

The problem is NP-hard and hence there is no known polynomial time algorithm to solve it. Different variants of the problems have been used in various fields, for example, the 1-DBP problem has applications in OR in the cutting stock problem and the machine scheduling problem, the 2-DBP problem has applications in memory allocation, and the 3-DBP problem has applications in logistics when objects have to be packed into containers for shipping or distribution.

In the field of logistics, while delivering items, real-world objects with different constraints need to be packed into containers, utilising the space of the container efficiently. For ease of packing and unpacking, it also needs to be ensured that the objects to be delivered first are near the opening of the container and the objects to be delivered last are towards the other end of the container. To address this need, we look at the 3-DBP problem and aim to provide a heuristic for the same.

Hence the problem statement of this project is: **Formulate, build, and test a heuristic to pack objects, each with its own weight, load bearing, and destination constraints, into a container of fixed dimensions with the aim of maximising the Volume Utilisation of the container.**

1.2 Previous Work

As the 3-DBP problem has numerous applications in the real world, there have been multiple approaches and attempts to provide optimal solutions.

In Theoretical Computer Science, the Bin Packing algorithm was first studied by Garey *et al.* (1972) in the context of memory allocation problems. They explore 4 heuristics: *First Fit*, *Best Fit*, *First Fit Decreasing*, and *Best Fit Decreasing* and provide bounds for their performance. Martello and Toth (1990) provide an LP formulation for the bin packing problem.

Bischoff and Ratcliff (1995) explore the different constraints that come up when developing an approach to a container loading program for different use cases. The paper suggests a layer-by-layer packing strategy when stability is the objective. When packing with a multi-drop constraint, they suggest a separation of the container space width-wise for different delivery locations.

Khan (2015) provides approximation algorithms for multidimensional bin packing. For the 2-DBP problem, a polynomial time algorithm with an asymptotic approximation ratio of $1 + \ln 1.5$ is provided.

Wang and Chen (2010) use a hybrid genetic algorithm for the 3-DBP.

Baltacioglu (2001) deals with the "Distributor's Pallet Packing Problem", where different boxes must be packed inside a minimum number of Pallets. The author discards weight and stability constraints, and employs a heuristic "modelling human intelligence".

Christensen and Rousøe (2009) explore solutions to the Container Loading Problem via greedy methods, and a tree search framework and tests results with real world-data from companies handling logistics and transportation.

Christensen *et al.* (2017) provide a survey of the available online and approximation algorithms for multidimensional bin packing. The survey discusses techniques used for bin packing, such as *Next Fit Decreasing Height* (seen in Coffman *et al.* (1980)), *Configuration LPs*, and *rounding objects to constant number of types*.

CHAPTER 2

Preliminaries

2.1 Assumptions

The following assumptions are made when packing:

- There is only one container, and one or more items are to be packed into the container
- All the items to be packed will be available at the start of packing, i.e. the packing is offline
- Each item has a delivery location associated with it. This location need not be unique, multiple items can be delivered in the same location.
- The Container has 3 dimensions and is cuboidal, it has 6 faces and all faces intersect at 90°
- All the items have 3 dimensions and are cuboidal, each item has 6 faces and all faces intersect at 90°
- The entirety of the container's dimensions are available for packing, and each item fully occupies its dimensions in space.
- All the dimensions in the problem are integers
- No two items can occupy the same location in space, i.e. no overlap is allowed
- We shall only pack an item atop a surface if all points in the surface below the box's location are of the same height, i.e. no overhang is allowed
- The item exerts pressure evenly across its base, i.e. the load due to the weight of the item is equal at all points across the item's bottom
- The packing is orthogonal, so rotation is only in increments of 90° with respect to all axes i.e. the items' surfaces will have to run parallel to the walls of the container when packed

2.2 Dataset

We use the OR Library: Beasley (1990) to obtain the "Container Loading with Weight Restrictions" dataset for testing. Among the 7 available files, `wtpack1` was used to test our heuristic.

`wtpack1` contains 100 problems, each with 3 different types of cuboidal items. Each container contains 3 dimensions, and each type of item contains:

- Weight of the item in kg
- Number of items of the type
- 3 Dimensions of the item - l, b, h in cm
- 3 Indicators specifying if each dimension can be in the vertical orientation $(0/1)$
- 3 Load-bearing limits of the item for when each dimension is vertical in $\mathrm{kg\,cm^{-2}}$

We add a randomly generated "destination" attribute (in the range 0 to 9) to each item to simulate the delivery order of the boxes and use a `Python` script to generate a separate file `wtpack1_i` for each problem i in `wtpack1`

All the problems in `wtpack1` have dimensions of $587 \times 233 \times 220\ \mathrm{cm^3}$ for their containers. All items have integral dimensions. The items have a mean dimension of $81.69 \times 57.75 \times 38.59\ \mathrm{cm^3}$ (mean obtained by taking the separate mean of each dimension).

Each problem has around $n = 150$ items to be packed into the container ($\mu_n = 150.44$). On average, the total volume of the items is 99.58% of the container volume. The total item volume never exceeds the container volume in the dataset and hence, there will always be enough "unpacked space" in the container for all the items (whether the spaces are usable is not guaranteed).

The dataset assigns weight to each item by multiplying its dimensions in m, i.e. it assumes the density of all items to be $1\,\mathrm{kg\,m^{-3}}$. We have used the default density of $1\,\mathrm{kg\,m^{-3}}$ for all items while testing. The script used to generate separate files also has the option to assign a randomly generated density (from a given range) to the items for further testing.

2.3 Classes

We define classes to store and easily process the input data. The purpose of each class, along with its members and their descriptions are provided in this section.

2.3.1 Location

This class represents the 3-D coordinates of an item. As seen in Figure 2.1, the coordinate of the rear left bottom of the container is set to $(0, 0, 0)$. The Z-axis coincides with the vertical edge of the container at the origin. Note that the axes are marked in orange and the remaining edges in blue.

Table 2.1: Members of the class *Location* and their descriptions

Member	Description
x	x-coordinate of the box
y	y-coordinate of the box
z	z-coordinate of the box

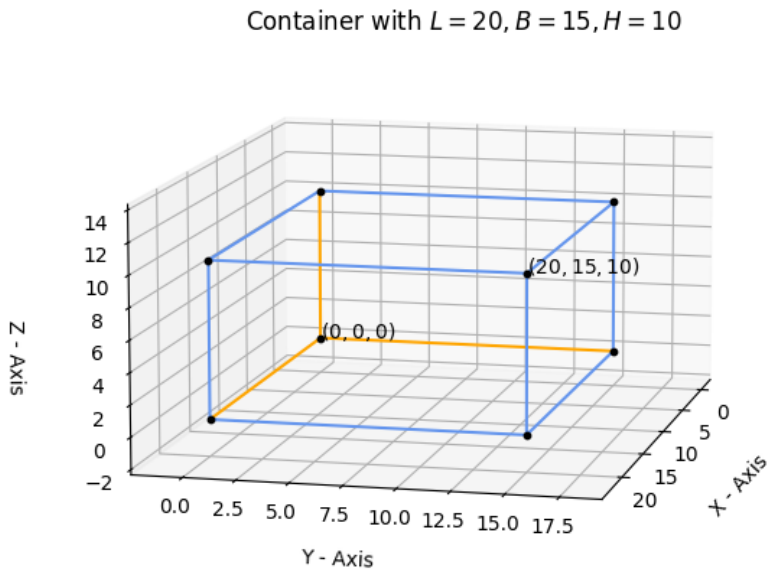

Figure 2.1: Origin and Axes of a Container

2.3.2 Package

This class represents the item that is being packed into the container.

Table 2.2: Members of the class *Package* and their descriptions

Member	Description
ID	Unique ID of the box
$dest$	Destination number of the box
wt	Weight of the box
$packed$	Boolean value to indicate if the box is packed
l	Length of the box
b	Breadth of the box
h	Height of the box
max_dim	Largest dimension of the box
l_1	Length of the container when packed
b_1	Breadth of the container when packed
h_1	Height of the container when packed
$orientation$	a list of size 3 containing a boolean indicator to show if l, b, h can be vertical respectively
$stack\_load$	a list of size 3 containing the load bearing ability of the box when l, b, h are vertical respectively
v_load	the load bearing ability of the box in its current orientation
pos	an object of class $Location$ containing the position of the box in the container
STRESS_LOAD()	function that returns load due to box $(\frac{wt}{l_1.b_1})$ in current orientation

2.3.3 Container

This class represents the container into which the items are being packed.

Table 2.3: Members of the class *Container* and their descriptions

Member	Description
L	Length of the container
B	Breadth of the container
H	Height of the container
h_grid	a 2D Array of size $L \times B$ containing the height to which each point is packed to all values initialised to 0
ld_lim	a 2D Array of size $L \times B$ containing the maximum further load allowed at each point all values initialised to ∞
$positions$	a set of coordinates of the bottom rear left corner of each available cuboidal space in the container
$packed\_items$	a list of objects of class $Package$ that have been packed in the container
VOL_OPT()	function that returns volume utilization of the container
FIT$(l, b, h, stress_load)$	function that takes dimensions of a box (l, b, h) the load due to its weight $stress\_load$ and returns the first position in $positions$ that can hold the box

2.3.4 Packer

This class represents the Packer that packs the $Package$ objects into the $Container$ object

Table 2.4: Members of the class *Packer* and their descriptions

Member	Description
$packages$	a list of objects of class $Packages$ that need to be packed
$container$	an object of class $Container$
PACK_ITEM(C, I)	function that takes a $Container$ object C and a $Package$ object I and packs I into C at the coordinates stored in $I.pos$
GREEDY_PACK$(C, items, start)$	function that takes a $Container$ object C, a list of $Package$ objects $items$, and an index $start$ and greedily packs packages from $items[start]$ to $items[0]$ into C
THREE_D_PACK$()$	function that uses tree search to pack $packages$ into $container$

2.4 Tree Search

In the proposed algorithm, we use tree search with "pruning of states" at each level to find better solutions with higher VU. We explain the exponential nature of the problem when running a direct tree search and the heuristic applied in this section.

2.4.1 Exponential number of states

Let us consider an initial empty state of the container. When we pack the first item in each of its 6 orientations at origin, this gives rise to 6 different states. This packing gives takes away one corner (origin) but gives rise to 3 new corners for possible packing. In general, when an item is packed, we get 3 new corners and lose 1 old corner, hence $c(n)$ - the maximum number of corners when the n^{th} is being packed is given by

$$c(n) = c(n-1) + 2 \tag{2.1}$$

Substituting c(1) = 1

$$c(n) = 2n - 1 \tag{2.2}$$

So, if we assume each item can be packed in a maximum of m orientations in each of the available corners, $s(n)$ - the maximum total number of possible states when the n^{th} item to be packed is given as:

$$s(n) = m \times c(n) \times s(n-1) \tag{2.3}$$

$$s(n) = m \times (2n-1) \times s(n-1) \tag{2.4}$$

$$s(n) = m^n . \prod_{k=1}^{n} (2k-1) \tag{2.5}$$

Substituting $s(0) = 1$ and $m = 6$ in (2.5), we get

$$s(n) = 6^n . \prod_{k=1}^{n} (2k-1) \tag{2.6}$$

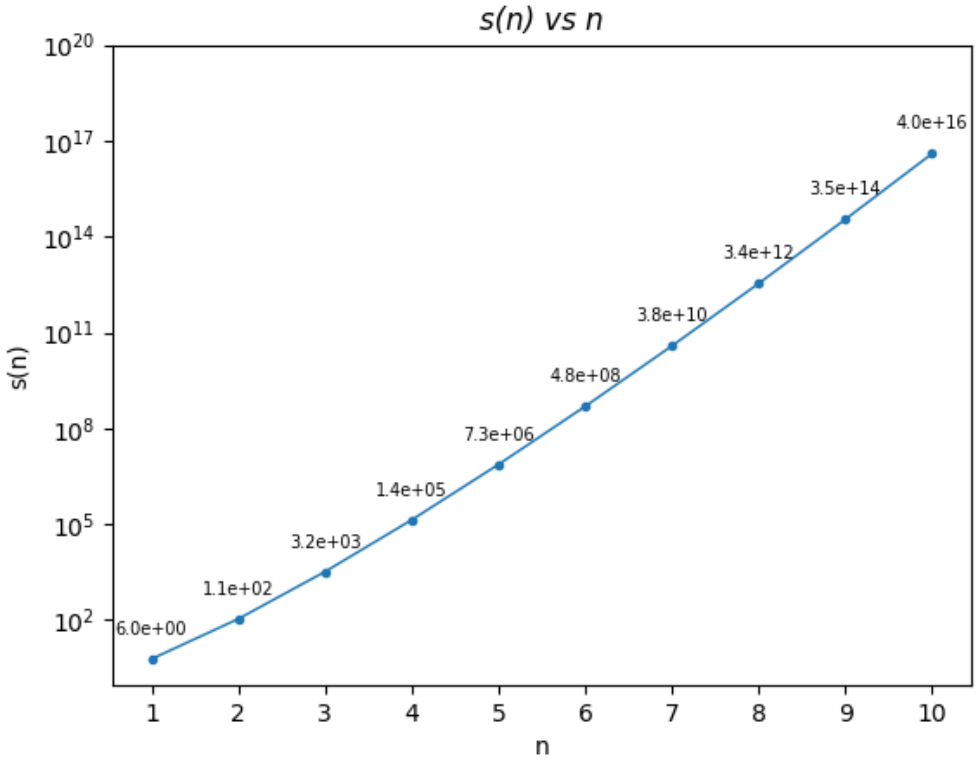

Figure 2.2: $s(n)$ plotted against n

In Figure 2.2, note that the scale of the y-axis denoting $s(n)$ is logarithmic, and that $s(n)$ rises exponentially, with

$$s(5) = 7\,348\,320$$
$$s(10) \approx 3.96 \times 10^{10}$$
$$s(15) \approx 2.91 \times 10^{27}$$

The time and memory memory constraints do not allow us to store all states possible states and iterate through them to find the best state (the approach is identical to brute forcing all combinatorial possibilities of packing). Hence we prune the tree after each item is packed as a heuristic.

2.4.2 Pruning the Tree

In order to cut down the runtime, and the memory requirements of the algorithm, at each level of the tree, we prune the nodes available and retain only the best t_w options available, where t_w is a predefined parameter called tree width.

In our context of packing, when packing n items, the i^{th} level of the tree contains states where items 1 to i have been iterated through. We define the "best" states as states which offer the highest VU when items $i + 1$ to n are packed greedily into that state. We pick the t_w best states and proceed to the next iteration.

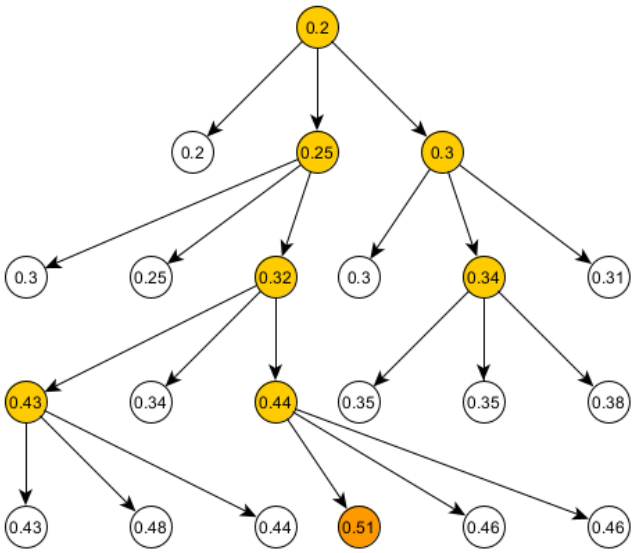

Figure 2.3: A tree search with $m = 3$, $t_w = 2$, $n = 4$

In Figure 2.3, a tree with $k = 3$, $t_w = 2$, $n = 4$ is illustrated. Let us consider that the value inside each node is its VU when packed greedily. The tree starts at n = 0, VU = greedily packing all items into the container (0.2 in this example). At each level, each state gives rise to 3 new states, we choose the best $t_w = 2$ states and proceed to the next level until we reach the final level, where we select the node with the highest value and end our search.

CHAPTER 3

Algorithm and Analysis

3.1 Algorithm

We use a tree search heuristic to pack the items into the container. A brief overview of the flow of the algorithm is given below:

- Data is read from input and a $Container$ object C and a list of $Package$ objects $items$ are created
- $items$ is sorted by its elements' destination, and then by their maximum load-bearing ability among its allowed orientations
- A $Packer$ object P is created with C and $items$ as its elements
- P.THREE_D_PACK() is invoked, and it invokes PACK_ITEM() and GREEDY_PACK() multiple times to pack C using Tree Search and returns the packed container

Note:

The ALLOWED_ORIENTATIONS(I) used in the algorithms takes a $Package$ object I return a list of size 6 where element i is $NULL$ if the i^{th} orientation is not permitted, or a $Package$ object I' where I has all the same values as I except its l_1, b_1, h_1 values are permuted among themselves (i.e. box is reoriented).

The $C.vol\_opt()$ used in the algorithms returns the VU of the container C. It adds the volumes of all the packed $Package$s and returns the ratio of the sum of packed volumes and the volume of the container C.

3.1.1 Algorithm 1 - FIT

This algorithm checks if an item can be fit into a Container. It is implemented as $Container$'s member function FIT(). The algorithm works as follows:

(i) We iterate through all available corners of cuboidal spaces in the Container (stored in $positions$) and check if the item can be placed at that corner

(ii) To check if the item can be placed, we check if the item when placed is within the confines of the container (line 9)

(iii) We then check if the base is of even height (line 14) and if it has sufficient load capacity (line 17) to support the item

(iv) If we find a coordinate that fits, we return the coordinate (line 31)

Algorithm 1 Checking if item fits in Container

```
1:  function FIT(self, l, b, h, stress_load)                ▷ self is Container object
2:      loc ← NULL
3:      for all p ∈ self.positions do
4:          pos_valid ← TRUE
5:          x ← p[0]
6:          y ← p[1]
7:          base ← self.h_grid[x, y]
8:          if x + l > self.L or y + b > self.B or base + h > self.H then
9:              continue                          ▷ checking if item is within limits
10:         end if
11:         for all 0 ≤ m < l do
12:             for all 0 ≤ n < b do
13:                 if self.h_grid[x + m, y + n] ≠ base then
14:                     pos_valid ← FALSE             ▷ checking if base is uniform
15:                     break
16:                 else if self.ld_lim[x + m, y + n] < stress_load then
17:                     pos_valid ← FALSE         ▷ checking if load is under limit
18:                     break
19:                 end if
20:             end for
21:             if pos_valid = FALSE then
22:                 break
23:             end if
24:         end for
25:         if pos_valid = TRUE then
26:             loc ← Location(x, y, base)                ▷ valid position found
27:             break
28:         end if
29:     end for
30:     return loc
31: end function
```

3.1.2 Algorithm 2 - PACK ITEM

This algorithm packs the item into the Container and updates the corresponding Package and Container variables. It is implemented as $Packer$'s member function PACK_ITEM(). The algorithm works as follows:

(i) We first check if the item has a valid position in the Container to be packed into (line 2)

(ii) If yes, we iterate across the dimensions of the container and update the grid at each point (m, n) as follows (lines 8 - 9):

- $h\_grid[m, n] = h\_grid[m, n] + h_1$ where h_1 is the height of the item
- $ld\_lim[m, n] = \text{MIN}(ld\_lim[m, n] - load, I.v\_load)$

  where,

  $load$: the load due to the weight of the item

  $I.v\_load$: the load-bearing ability of the item

  We update ld_lim as given above because ld_lim is limited by the minimum of the load-bearing ability of the item ($I.v\_load$) and the updated load-bearing ability at that point, which is the previous load bearing ability (ld_lim) reduced by the load due the item's weight when packed ($load$).

(iii) We add the newly created corners to the $positions$ set of the container (line 14).

(iv) We set the $packed$ flag in the item to TRUE and append the package to the container's list of packed items (line 20)

Algorithm 2 Packing item into Container

Require: C is a $Container$ Object, I is a $Package$ object

1: **function** PACK_ITEM(self, C, I) ▷ $self$ is $Packer$ object
2: **if** $I.pos$ is $NULL$ **then**
3: **return**
4: **end if**
5: $load \leftarrow I.stress\_load()$
6: **for all** $(I.pos.x \leq m < I.pos.x + I.l_1)$ **do**
7: **for all** $(I.pos.y \leq n < I.pos.y + I.b_1)$ **do**
8: $C.h\_grid[m, n] \leftarrow C.h\_grid[m, n] + I.h_1$ ▷ updating height grid
9: $C.ld\_lim[m, n] \leftarrow \text{MIN}(C.ld\_lim[m, n] - load, I.v\_load)$
10: ▷ updating load limit
11: **end for**
12: **end for**
13: **if** $I.pos.x + I.l_1 < C.L$ **then**
14: $C.positions.\text{ADD}(I.pos.x + I.l_1, I.pos.y)$
15: **end if**
16: **if** $I.pos.y + I.b_1 < C.B$ **then**
17: $C.positions.\text{ADD}(I.pos.x, I.pos.y + I.b_1)$
18: **end if** ▷ adding new corners created after packing to positions
19: $I.packed \leftarrow \text{TRUE}$
20: $C.packed\_items.\text{APPEND}(I)$
21: **return** C
22: **end function**

3.1.3 Algorithm 3 - GREEDY PACK

This algorithm greedily packs the items from index $start$ to 0 in the list $items$. It is implemented as $Packer$'s member function GREEDY_PACK(). The algorithm works as follows:

(i) First, we make a copy C_1 of the container C (line 2)

(ii) Then we iterate the $items$ list from index $start$ to 0 (line 3)

(iii) And for each item, we generate its allowed orientations and check for each valid orientation if it fits via the FIT() function (line 11)

(iv) If an orientation is valid and can be packed into the container, the function greedily packs the item into the container and exits the iteration (line 13)

(v) The function finally returns the VU after greedy packing is completed (line 18)

Algorithm 3 Greedily packing Packages into Container

Require: C is a $Container$ Object, $items$ is a list of $Package$ object, $start \geq 0$

```
1: function GREEDY_PACK(self, C, items, start)          ▷ self is Packer object
2:     C_1 ← C
3:     for (k = start; k ≥ 0; k ← k − 1) do
4:         I ← items[i]
5:         Iarr ← ALLOWED_ORIENTATIONS(I)
6:         for all 0 ≤ j < 6 do:
7:             if Iarr[j] is NULL then
8:                 continue
9:             end if
10:            I_1 ← Iarr[j]
11:            I_1.pos ← C_1.FIT(I_1.l_1, I_1.b_1, I_1.h_1, I_1.STRESS_LOAD())
12:            if I_1.pos is not NULL then
13:                C_1 ← self.PACK_ITEM(C, I_1)
14:                break
15:            end if
16:        end for
17:    end for
18:    return C_1.VOL_OPT()
19: end function
```

3.1.4 Algorithm 4 - THREE D PACK

This algorithm performs tree search to maximise the VU of the Container. It is implemented as $Packer$'s member function THREE_D_ITEM(). The algorithm works as follows:

(i) We first initialise an empty list $states$, each element in $states$ is a tuple of a $Container$ object and a floating value. (line 2)

(ii) We initialise $states$ with an empty Container and its GREEDY_PACK() VU (line 4)

(iii) We then iterate across $items$ in reverse (line 5). As $items$ is sorted in ascending order of delivery points, this ensures that the packages that will be delivered last are packed first in the innermost corners available

(iv) For each I in $items$:
   (a) We iterate across all the states available in reverse order
   (b) In each iteration, we first make a copy of the current state, find the volume utilization when we greedily pack all remaining packages except the current package I and append the new state and its volume utilization to $states$ (lines 9-11)
   (c) We then iterate across all valid orientations of I. In each iteration we make a copy of the current state and try to pack in the current orientation of I (line 12)
   (d) if I's current orientation can be packed, we pack that into the new state and we find the volume utilization when we greedily pack all remaining packages and append the new state and its volume utilization to $states$ (line 22)
   (e) As the greedy packing from the current state must
      - either not fit I
      - or fit I in one of its valid orientations

      The same state must have been generated in one of the steps iv.a & iv.c. So we remove the current state from $states$ to avoid redundant states (line 26)

(v) We sort $states$ by its volume utilization and retain only the top $Tree\_Width$ states. This is the tree search part of the algorithm. (lines 28-29)

(vi) After all iterations, the state with the highest VU is returned (line 31)

Algorithm 4 Tree Search for highest VU of Container

Require: C is a $Container$ Object, $items$ is a list of $Package$ object, $Tree\_Width$ a predefined constant

function THREE_D_PACK(self, C, items) ▷ $self$ is $Packer$ object
  $states \leftarrow [\,]$ ▷ $states$ is a list of $[Container, float]$
  $g \leftarrow self.\text{GREEDY\_PACK}(C, items, \text{LEN}(items) - 1)$
  $states.\text{APPEND}([C, g])$
  for I in $\text{REVERSE}(items)$ **do**
    $I \leftarrow items[i]$
    $Iarr \leftarrow \text{ALLOWED\_ORIENTATIONS}(I)$
    for $(k = \text{LEN}(states);\ k \geq 0;\ k \leftarrow k - 1)$ **do**
      $C_1 \leftarrow states[k][0]$
      $g \leftarrow self.\text{GREEDY\_PACK}(C_1, items, i - 1)$
      $states.\text{APPEND}([C_1, g])$
      for all $0 \leq j < 6$ **do**:
        if then$Iarr[j]$ is $NULL$
          continue
        end if
        $C_1 \leftarrow states[k][0]$
        $I_1 \leftarrow Iarr[j]$
        $I_1.pos \leftarrow C_1.\text{FIT}(I_1.l_1, I_1.b_1, I_1.h_1, I_1.\text{STRESS\_LOAD}())$
        if $I_1.pos$ is not $NULL$ **then**
          $C_1 \leftarrow self.\text{PACK\_ITEM}(C, I_1)$
          $g \leftarrow self.\text{GREEDY\_PACK}(C_1, items, i - 1)$
          $states.\text{APPEND}([C_1, g])$
          break
        end if
      end for
      erase $states[k]$
    end for
    $states \leftarrow \text{SORT}(states)$
    $states \leftarrow \text{RESIZE}(states, Tree\_Width)$
  end for
  return $states[0][0]$
end function

3.2 Analysis

We look at the time complexities of the algorithms given in section 3.1

3.2.1 Time Complexity of Algorithm 1 - FIT

The algorithm iterates through all positions and then checks if the position is valid across the dimensions of the item. The operations within the iterations themselves are all simple $O(1)$ operations.

Time Complexity of FIT : $\mathcal{O}(\text{LEN}(positions) \times l \times b)$

We know that the $\text{LEN}(positions)$ = no. of rear bottom left corners of cuboidal spaces in the container. By defining the number of items as n and by eqn. (2.2) we get,

$$\text{LEN}(positions) \leq c(n) = 2n - 1$$

Hence $\text{LEN}(positions) \leq k \cdot n$, $k \in \mathbb{N}$. By substituting that and by defining the maximum dimension of a package as d_m, we get

$$\mathbf{T}_{fit} = \mathcal{O}(n \times l \times b) \tag{3.1}$$

$$\implies \mathbf{T}_{fit} = \mathcal{O}(n \cdot d_m^2) \tag{3.2}$$

3.2.2 Time Complexity of Algorithm 2 - PACK ITEM

The algorithm iterates across the dimensions of the package. The operations within the iterations, and other external operations are all simple $O(1)$ operations. By defining the maximum dimension of a package as d_m, we get

$$\mathbf{T}_{pack} = \mathcal{O}(I.l \times I.b) \tag{3.3}$$

$$\implies \mathbf{T}_{pack} = \mathcal{O}(d_m^2) \tag{3.4}$$

3.2.3 Time Complexity of Algorithm 3 - GREEDY PACK

The algorithm iterates from $start$ to 0, and inside each iteration, 6 orientations of the item are iterated through. For each iteration FIT() and PACK_ITEM() are invoked. The operations within the iterations, and other external operations are all simple $O(1)$ operations.

As start can take a maximum of LEN$(items)$, by defining LEN$(items) = n$ we get

$$\mathbf{T}_g = \mathcal{O}((start \times 6) \times (\mathbf{T}_{fit} + \mathbf{T}_{pack})) \tag{3.5}$$

Substituting from (3.2) and (3.4), we get

$$\mathbf{T}_g = \mathcal{O}((start \times 6) \times (n \times d_m^2 + d_m^2))) \tag{3.6}$$

$$\mathbf{T}_g = \mathcal{O}((start \times 6) \times (n \times d_m^2)) \tag{3.7}$$

By applying $start \leq n$ and removing the constant, we get

$$\implies \mathbf{T}_g = \mathcal{O}(n^2 \cdot d_m^2) \tag{3.8}$$

3.2.4 Time Complexity of Algorithm 4 - THREE D PACK

The algorithm first invokes the greedy algorithm. Then it iterates through items. In each item, it iterates through all the available states, and for each state, it goes through each of its 6 orientations.

For each orientation, FIT, PACK_ITEM, and GREEDY_PACK is invoked, and the list $states$ is sorted.

Let us define the number of items as n and the maximum item dimension as d_m. Let us take the time to sort $states$ as T_{sort}. Now,

$$\text{LEN}(items) = n$$

$$\text{LEN}(states) \leq k \cdot t_w \qquad (t_w = \text{Tree width}, k \in \mathbb{N})$$

And,

$$\mathbf{T}_{sort} = \mathcal{O}(\text{LEN}(states) \log \text{LEN}(states))$$
$$\implies \mathbf{T}_{sort} = \mathcal{O}(t_w \log t_w)$$

Substituting the above inequalities, we get

$$\mathbf{T}_{3D} = \mathbf{T}_g + (\text{LEN}(items) \times (\text{LEN}(states) \times 6 \times (\mathbf{T}_{fit} + \mathbf{T}_{pack} + \mathbf{T}_g) + \mathbf{T}_{sort})) \tag{3.9}$$

Substituting from (3.2), (3.4), and (3.8), we get

$$\mathbf{T}_{3D} = \mathcal{O}(n^2 d_m^2 + (n \times (t_w \times 6 \times (n.d_m^2 + d_m^2 + n^2.d_m^2) + t_w \log t_w))) \tag{3.10}$$
$$\mathbf{T}_{3D} = \mathcal{O}(n \times (6.t_w.n^2 d_m^2 + t_w \log t_w)) \tag{3.11}$$

As $\log t_w \ll n^2 d_m^2$

$$\mathbf{T}_{3D} = \mathcal{O}(n \times (6.t_w.n^2 d_m^2)) \tag{3.12}$$

Removing the constant, we get

$$\implies \mathbf{T}_{3D} = \mathcal{O}(t_w \cdot n^3 \cdot d_m^2) \tag{3.13}$$

The total time complexity of the entire packing algorithm THREE D PACK is given in (3.13)

Where,

t_w - Tree Width

n - number of items to be packed

d_m - the maximum dimension of an item

3.3 Parallelization

To speed up the given algorithm, we propose the following parallelization.

In the FIT algorithm, we iterate across all the points $(x + m, y + n)$ for m, n in the range $0 \leq m < l$, $0 \leq n < b$ in lines 12 - 13 and check elementwise if they satisfy a certain binary condition.

Similarly, in the PACK_ITEM algorithm, we iterate across all the points $(x+m, y+n)$ for m, n in the range $I.pos.x \leq m < I.pos.x + I.l_1$, $I.pos.y \leq n < I.pos.y + I.b_1$ in lines 8 - 10 and update the values elementwise.

Assuming in ideal conditions, that the elementwise check and update is sped up to some $\mathcal{O}(1)$ constant from the current $\mathcal{O}(d_m^2)$, our existing algorithm will be sped up by a factor of d_m^2 with an updated time complexity of

$$T_{upd} = \mathcal{O}(t_w.n^3)$$

An implementation of this parallelization was performed using the `CuPy` library, that runs `Python` code written with `NumPy` on GPU via plugging into `CUDA`. The GPU version of the code, while producing the same results, is slower. Further results with additional details are given in section 4.4.

CHAPTER 4

Results

4.1 Volume Utilization by Algorithm

The above heuristic was implemented in `Python3` and tested with `wtpack1` from the OR Library as mentioned in Section 2.2. The tree width t_w was set to 5 in all test cases.

Table 4.1: VU Results of `wtpack1`

Problem Nos.	Mean VU	Max VU	Min VU	SD of VU	Mean time (in s)
1-10	0.596	0.795	0.460	0.101	613.442
11-20	0.630	0.774	0.480	0.099	558.061
21-30	0.571	0.781	0.000	0.216	250.870
31-40	0.619	0.804	0.308	0.153	784.924
41-50	0.568	0.822	0.214	0.165	398.914
51-60	0.509	0.772	0.000	0.233	375.627
61-70	0.505	0.764	0.000	0.212	803.659
71-80	0.590	0.778	0.384	0.120	633.037
81-90	0.571	0.702	0.000	0.197	332.097
91-100	0.624	0.765	0.464	0.084	737.944
Total	0.578	0.822	0.000	0.171	548.858

The zeroes appear in the Min VU column because Problems $30, 56, 65, 85$ had due very high runtime due high number of items $(n > 300)$ and were skipped during testing. Removing these outlier Problems, our results are as seen in Table 4.2

Table 4.2: VU Results of `wtpack1` with outliers removed

Problem Nos.	Mean VU	Max VU	Min VU	SD of VU	Mean time (in s)
1-10	0.596	0.795	0.460	0.101	571.727
11-20	0.630	0.774	0.480	0.099	571.727
21-30	0.633	0.781	0.412	0.102	571.727
31-40	0.622	0.804	0.308	0.153	571.727
41-50	0.580	0.822	0.214	0.174	571.727
51-60	0.532	0.709	0.209	0.149	571.727
61-70	0.601	0.778	0.385	0.136	571.727
71-80	0.570	0.747	0.384	0.100	571.727
81-90	0.638	0.765	0.464	0.081	571.727
91-96	0.635	0.715	0.537	0.062	571.727
Total	0.602	0.822	0.209	0.127	571.727

Even though we only optimized for maximising VU, we can also analyse the corresponding ratio of items packed to items in the input. We shall call the ratio of Packed to Input item count as PIR.

Table 4.3: PIR Results of `wtpack1`

Problem Nos.	Mean PIR	Max PIR	Min PIR	SD of PIR	Mean time (in s)
1-10	0.646	0.891	0.480	0.120	613.442
11-20	0.674	0.846	0.596	0.096	558.061
21-30	0.596	0.829	0.000	0.230	250.870
31-40	0.675	0.839	0.350	0.136	784.924
41-50	0.631	0.826	0.225	0.166	398.914
51-60	0.556	0.837	0.000	0.243	375.627
61-70	0.532	0.813	0.000	0.229	803.659
71-80	0.652	0.818	0.438	0.120	633.037
81-90	0.593	0.814	0.000	0.212	332.097
91-100	0.689	0.830	0.549	0.086	737.944
Total	0.624	0.891	0.000	0.181	548.858

Table 4.4: PIR Results of `wtpack1` with outliers removed

Problem Nos.	Mean PIR	Max PIR	Min PIR	SD of PIR	Mean time (in s)
1-10	0.646	0.891	0.480	0.120	613.442
11-20	0.674	0.846	0.596	0.096	558.061
21-30	0.658	0.829	0.427	0.116	290.733
31-40	0.692	0.839	0.350	0.138	770.719
41-50	0.636	0.837	0.225	0.171	402.907
51-60	0.570	0.793	0.288	0.145	398.153
61-70	0.643	0.813	0.385	0.149	831.247
71-80	0.617	0.818	0.438	0.107	709.177
81-90	0.694	0.830	0.549	0.085	581.541
91-96	0.689	0.799	0.551	0.073	554.325
Total	0.650	0.891	0.225	0.131	571.727

Similar to VU, we display the PIR results after removing the results in Table 4.4

The mean and standard deviation of VU and PIR obtained by the proposed heuristic is given below:

$$\mu_{\text{VU}} = 0.602$$

$$\sigma_{\text{VU}} = 0.127$$

$$\mu_{\text{PIR}} = 0.650$$

$$\sigma_{\text{PIR}} = 0.131$$

4.2 Runtime vs Number of Items

As we see in Figure 4.1, the running time $\mathbf{T}_{3D}$ generally increases with the increase in the number of items to be packed n.

The tree width is constant across all tests, and the maximum item dimension (d_m) varies within a limited range (μ_{d_m} = 85.78, σ_{d_m} = 19.11), so n is the main contributor to deciding $\mathbf{T}_{3D}$.

When we calculate the best fitting cubic ax^3+b using the `scipy` package's `curve_fit()`, we get the polynomial:

$$\text{BEST\_FIT}(x) \approx (1.22 \times 10^{-4}) \cdot x^3 + (1.18 \times 10^2)$$

We can see in Figure 4.1 that the curve captures the trend in the data and that this is in accordance with our time complexity of $\mathcal{O}(t_w \cdot n^3 \cdot d_m^2)$ where $\mathbf{T}_{3D} \propto n^3$

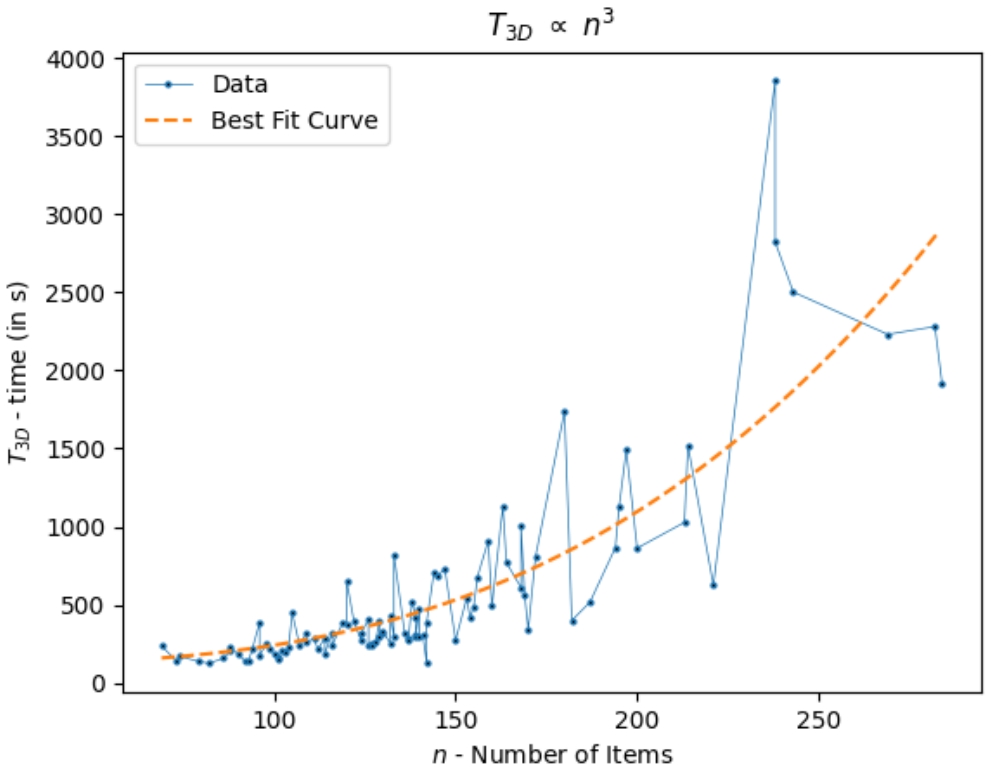

Figure 4.1: No. of items (n) vs Time to run THREE_D_PACK ($\mathbf{T}_{3D}$)

4.3 Visualisation of Packing

To visualise the packing, a visualiser was built in `Python3` using `matplotlib`. The script takes the `json` file output by the Packer script and produces a visualisation of the results.

The packing for the demo input produced by the Packer and Plotted by the visualiser is given in Figure 4.2

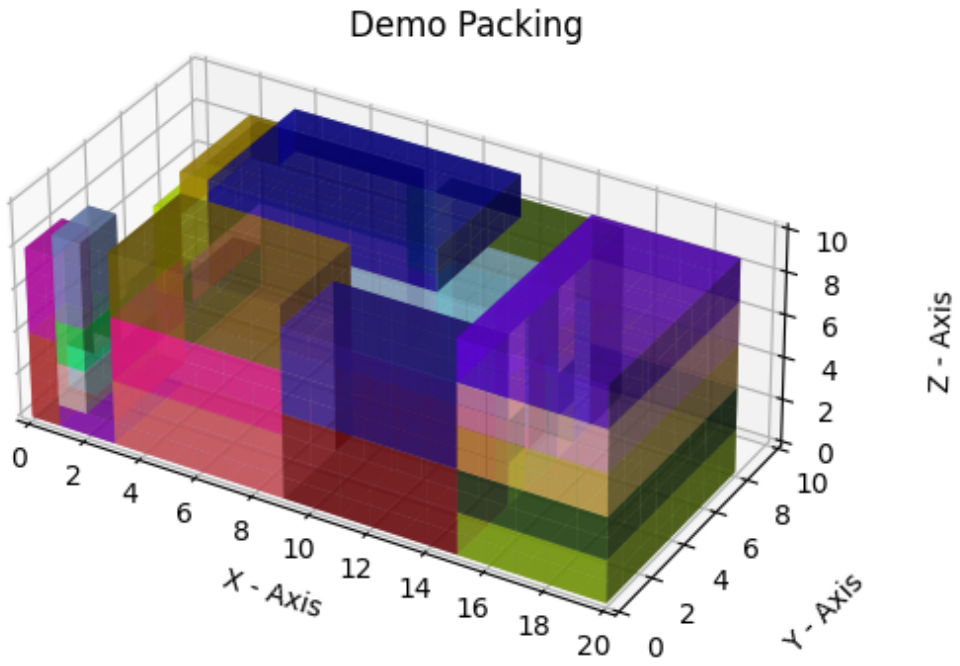

Figure 4.2: Packing of Demo Input

The Demo Input's container is of dimensions $20 \times 10 \times 10\ \mathrm{cm}^3$, and the items are as follows:

- $8 \times 5 \times 2\ \mathrm{cm}^3$ - 10 Items
- $6 \times 4 \times 3\ \mathrm{cm}^3$ - 15 Items
- $4 \times 2 \times 1\ \mathrm{cm}^3$ - 15 Items

Similar to `wtpack`, all items were assumed to have a density of $1\ \mathrm{kg\,m}^{-3}$. And the same load bearing capacities from `wtpack1_0` were assigned to each box. VU of 0.768 and PIR of 0.850 was obtained.

4.4 Runtime with Parallelisation

To test the performance of the GPU code with respect to the CPU code, 5 problems from `wtpack1` were randomly picked and their respective runtimes were compared. The respective time taken is given in Figure 4.3

The GPU code takes 5 to 10 times longer than the CPU when packing the same problem (mean among sample = 7.44).

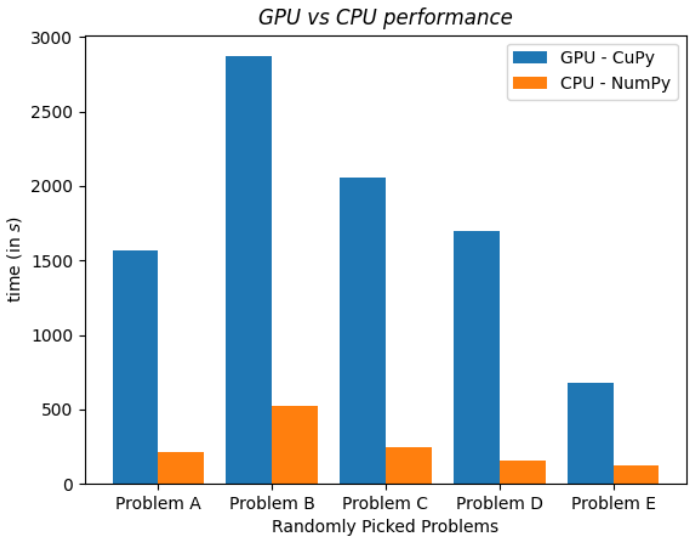

Figure 4.3: Time taken to pack the same randomly picked problems from `wtpack1` with and without parallelization

This increase in runtime might be due to the overhead from copying the data grids from CPU memory to GPU memory, or the inefficiency of optimisation provided by the `CuPy` library. A direct plug-in to `CUDA` might help uncover the potential speed-up.

CHAPTER 5

Future Work

Additional Constraints:

The stability of the container can be subject to tighter constraints by changing the initial values of ld_lim as needed - points above fragile points in the container, such as right above the axles in a truck can be initialised to a low ld_lim value. Different constraints for different use cases, as given in Bischoff and Ratcliff (1995) can be added to this heuristic for case-appropriate packing.

Optimisation Metrics:

The metric for optimisation will vary for each use case. In this heuristic, we aimed to maximise the VU of the Container, we can also choose to optimize PIR. For real-world application, if the cost and the revenue for each successful delivery is available, we can also maximise the net profit generated from packing and delivering the items.

Parallelization Speed-up:

The speed-up that can theoretically be obtained by parallelizing the grid update and check operations (in Section 3.3) was not realised in the current implementation (in Section 4.4). Further testing by directly implementing the heuristic in `CUDA` and comparing it with a `C++` implementation might reveal the potential speedup possible due to parallelization.

APPENDIX A

Python Implementation

The `Python` implementation for the heuristic mentioned above is available at this GitHub Repository.

The repository has three repository has the following 3 branches at the time of writing:

- `main` - This branch implements the heuristic using the `NumPy` library. It is currently the fastest implementation of the heuristic
- `basic` - This branch implements the heuristic using just 2-D built-in `Python` lists. It is implemented with the least system requirements but is slower than `main`
- `gpu` - This branch implements the heuristic using the `CuPy` library, it has been tested with `CUDA 12.1`. It is mainly experimental and is the slowest among all branches.

To test and run code on a local system, the steps are as follows:

```
$ git clone https://github.com/nanthamanish/WeightedPacking.git
$ python make_inputs.py <wtpackX>
```

To run Packer on one file, run :

```
$ python main.py <wtpackX_Y> <output_file>
```

To run Packer on all input files, run:

```
$ python test_all.py <wpackX> <output_file>
```

Here, $0 \leq X \leq 7$, $0 \leq Y \leq 99$, and `output_file` is to be specified without a file extension.

REFERENCES

1. **Baltacioglu, E.** (2001). *The Distributer's Three-Dimensional Paller-Packing Problem: A Human Intelligence-Based Heuristic Approach*. Ph.D. thesis, Graduate School of Engineering and Management, Air Force Institute of Technology.

2. **Beasley, J. E.** (1990). Or-library: Distributing test problems by electronic mail. *The Journal of the Operational Research Society*, **41**(11), 1069–1072. ISSN 01605682, 14769360. URL http://www.jstor.org/stable/2582903.

3. **Bischoff, E.** and **M. Ratcliff** (1995). Issues in the development of approaches to container loading. *Omega*, **23**(4), 377–390. ISSN 0305-0483. URL https://www.sciencedirect.com/science/article/pii/030504839500015G.

4. **Christensen, H. I.**, **A. Khan**, **S. Pokutta**, and **P. Tetali** (2017). Approximation and online algorithms for multidimensional bin packing: A survey. *Computer Science Review*, **24**, 63–79. ISSN 1574-0137. URL https://www.sciencedirect.com/science/article/pii/S1574013716301356.

5. **Christensen, S. G.** and **D. M. Rousøe** (2009). Container loading with multi-drop constraints. *International Transactions in Operational Research*, **16**(6), 727–743. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1475-3995.2009.00714.x.

6. **Coffman, E. G., Jr.**, **M. R. Garey**, **D. S. Johnson**, and **R. E. Tarjan** (1980). Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM J. Comput.*, **9**(4), 808–826. ISSN 0097-5397. URL https://doi.org/10.1137/0209062.

7. **Garey, M. R.**, **R. L. Graham**, and **J. D. Ullman**, Worst-case analysis of memory allocation algorithms. *In Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72. Association for Computing Machinery, New York, NY, USA, 1972. ISBN 9781450374576. URL https://doi.org/10.1145/800152.804907.

8. **Khan, A.** (2015). *Approximation Algorithms for Multidimensional Bin Packing*. Ph.D. thesis, School of Computer Science, Georgia Institute of Technology.

9. **Martello, S.** and **P. Toth**, *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley Sons, Inc., USA, 1990. ISBN 0471924202.

10. **Wang, H.** and **Y. Chen**, A hybrid genetic algorithm for 3d bin packing problems. *In 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*. 2010.