POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Title

Tesi di Laurea Magistrale in
Xxxxxxx Engineering - Ingegneria Xxxxxxx

Author: **Name Surname**

Student ID: 000000
Advisor: Prof. Name Surname
Co-advisors: Name Surname, Name Surname
Academic Year: 20XX-XX

Abstract

Here goes the Abstract in English of your thesis followed by a list of keywords. The Abstract is a concise summary of the content of the thesis (single page of text) and a guide to the most important contributions included in your thesis. The Abstract is the very last thing you write. It should be a self-contained text and should be clear to someone who hasn't (yet) read the whole manuscript. The Abstract should contain the answers to the main scientific questions that have been addressed in your thesis. It needs to summarize the adopted motivations and the adopted methodological approach as well as the findings of your work and their relevance and impact. The Abstract is the part appearing in the record of your thesis inside POLITesi, the Digital Archive of PhD and Master Theses (Laurea Magistrale) of Politecnico di Milano. The Abstract will be followed by a list of four to six keywords. Keywords are a tool to help indexers and search engines to find relevant documents. To be relevant and effective, keywords must be chosen carefully. They should represent the content of your work and be specific to your field or sub-field. Keywords may be a single word or two to four words.

Keywords: here, the keywords, of your thesis

Abstract in lingua italiana

Qui va l'Abstract in lingua italiana della tesi seguito dalla lista di parole chiave.

Parole chiave: qui, vanno, le parole chiave, della tesi

Contents

1 | Introduction

Intro

Case study

Overview

Static stability

2 | Literature review

3 | Problem description and mathematical formulation

3.1. 3D Bin Packing Problem

3.2. Support

3.3. MILP Formulation

Conceptual model

A conceptual model of the problem we are trying to solve would be:

minimize	unused volume in used bins
subject to	all items assigned to one and only one bin
	all items within the bin dimensions
	no overlaps between items in the same bin
	all items with support

We can now provide the formal definition of the 3DBPP by formulating a mixed integer linear programming problem model.

Formal model

We'll now introduce a MILP model for the standard 3DBPP problem definition and then we'll expand it to address the stability constraint afterwards.

We start by defining the known sets and parameters of the problem.

Sets

$$
\begin{aligned}
I = \{1, \ldots, n\} : &\quad \text{set of items} \\
B = \{1, \ldots, m\} : &\quad \text{set of bins}
\end{aligned}
$$

Parameters

$$
\begin{aligned}
W \times D \times H &\quad \text{width} \times \text{depth} \times \text{height of a bin} \\
V &\quad \text{bin volume} \\
w_i \times d_i \times h_i &\quad \text{width} \times \text{depth} \times \text{height of item } i \qquad \forall i \in I \qquad (3.1)
\end{aligned}
$$

Variables We can now introduce the following sets of integer variables

$$
\begin{aligned}
(x_i, y_i, z_i) &\quad \text{bottom front left corner of an item} && \forall i \in I \qquad (3.2) \\
(x'_i, y'_i) &\quad \text{back right corner of an item} && \forall i \in I \qquad (3.3) \\
r_i &\quad \begin{cases} 1, \text{if item } i \text{ is rotated } 90° \text{ over its z-axis} \\ 0, \text{otherwise} \end{cases} && \forall i \in I \qquad (3.4) \\
u_{ib} &\quad \begin{cases} 1, \text{if item } i \text{ is placed in bin } b \\ 0, \text{otherwise} \end{cases} && \forall i \in I, \forall b \in B \\
x^p_{ij} &\quad \begin{cases} 1, \text{if } x_i \leq x'_j \\ 0, \text{otherwise} \end{cases} && \forall i, j \in I \\
y^p_{ij} &\quad \begin{cases} 1, \text{if } y_i \leq y'_j \\ 0, \text{otherwise} \end{cases} && \forall i, j \in I \\
z^p_{ij} &\quad \begin{cases} 1, \text{if } z_i \leq z_j + h_j \\ 0, \text{otherwise} \end{cases} && \forall i, j \in I \\
z^{\max}_b &\quad \text{maximum height of bin } b && \forall b \in B
\end{aligned}
$$

Given a coordinate system, each item i can be represented univocally in 3D space by eqs. (3.1) to (3.4) as seen in figure 3.1

Static stability constraints

We now extend the model introduced in section 3.3 to introduce constraints addressing static stability.

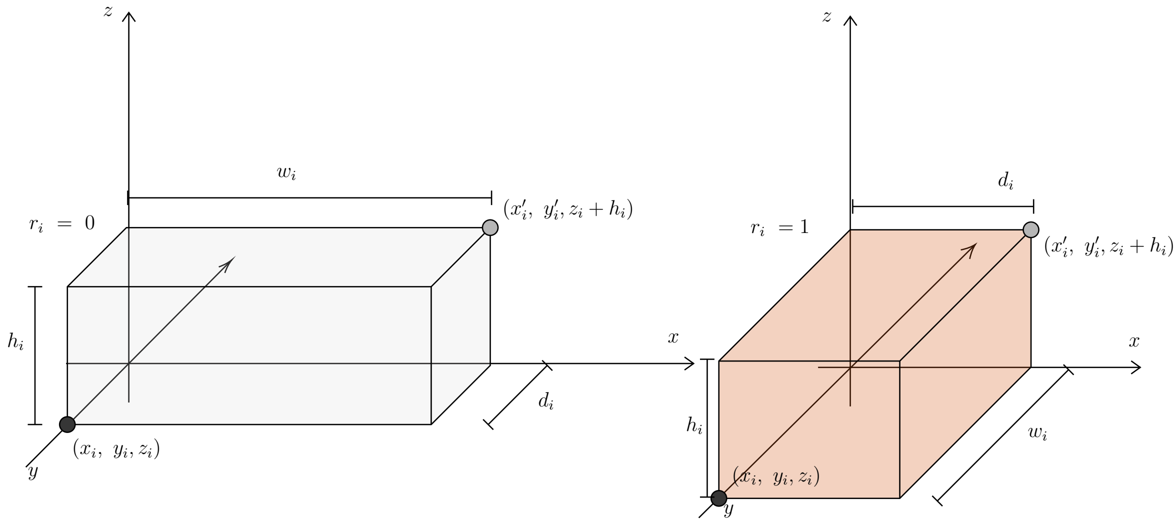

Figure 3.1: Coordinate system representation for a generic item i given its rotation r_i

Additional Parameters

α_s : support area threshold

β_s : support height tolerance

4 | Solution algorithms

In this chapter, we describe a solution to the 3D bin packing problem with static stability. A solution candidate to the problem can be found by conducting a search over the tree of possible packings or states. In section 4.1 we describe what a state or packing is and its representation. Since an exhaustive search isn't feasible, a heuristic search is conducted by combining a beam search algorithm described in section 4.2 and a constructive heuristic described in section 4.3. The proposed algorithm takes in input an initial feasible state (as defined in section 4.1.2) usually represented by the empty state (4.3) and outputs the best scoring state based on an ordering function defined in section 4.2.1.

4.1. State

States or packings are partial solutions to the 3DBPP. Given the formal definition of the problem (3.3) a few new definitions are introduced to facilitate the algorithm's definition.

Definition 4.1 (Unpacked item). *An item $i \in I$ is unpacked* ***iff***

$$\sum_{b \in B} u_{ib} = 0$$

It is also assumed that variables identifying an item's position, and rotation are independent between states (changes to their values in state s won't affect state s').

A state s can then be defined as follows

- U: the set of unpacked items
- B: the set of bins
- $Q = (q_1, q_2, \dots, q_b)$: the set of supporting structures for each bin $b \in B$
- p: the insertion pending on this state (described by def. 4.4)

Observation 4.1. *Given two states s and s' we can have that $|s.B| \neq |s'.B|$ since the number of bins is also a variable in the proposed heuristic*

We can also trivially define a function that determines if a state is a final state

Definition 4.2. *A state s is final if there are no more items to pack*

$$IsFinal(s) = \begin{cases} 1, & s.U = \emptyset \\ 0, & otherwise \end{cases} \tag{4.1}$$

Each bin b has additional data that is contained in a structure $q_b \in s.Q$ used to facilitate the execution of the algorithm.

Let us introduce the concept of packed items inside a bin:

Definition 4.3 (Packed item)**.** *Given a state s and a bin $b \in s.B$, we say that item*

$$i \in I \;\; \textit{is packed in} \;\; b \;\; \textbf{\textit{iff}} \quad u_{ib} = 1$$

Given a bin $b \in s.B$ we can then define structure q_b as follows

- J: the set of items that are packed inside b
- Z: the set of planes inside b (section 4.3)
- T: the AABB Tree (section 4.1.1) representing the items inside b

Notice that two separate sets containing the items packed in b are present inside q_b but adding and accessing items in $q_b.J$ has a time complexity of $O(1)$ given an underlying implementation as HashSet while maintaining $q_b.T$ usually has a time complexity of $O(log(|q_b.J|))$.

The reason to include an AABB Tree inside this structure is further explained in sections 4.1.2 and 4.3.1

4.1.1. AABB Tree

In order to determine the feasibility of a given state, a way of checking for overlaps with items already placed is needed. Since our formulation of the problem only allows for 90 deg rotations over the z-axis. Every item in a solution, by the problem formulation (3.1), is contained inside a bounding box and this box is axis-aligned. An adequate structure to compute overlaps is then an Axis-Aligned Bounding Box Tree (AABB Tree) [1].

AABB Trees are bounding volume hierarchies typically used for fast collision detection and they usually offer a few operations:

- $AABBInsert(i)$: which allows inserting an axis-aligned box i in the tree
- $AABBOverlaps(i)$: which allows determining if an axis-aligned box i overlaps an element in the tree
- $AABBClosest(i, d)$: which, given an axis-aligned box i and a direction $d \in \{XP, XN, YP, YN, ZP, ZN\}$ along an axis, returns the closest element inside the tree following that direction starting from the box i

If the tree is properly balanced each operation on average has a time complexity of $O(log(n))$ where n is the number of elements in the tree. Maintaining an AABB Tree in the state allows us to do checks for feasibility during the construction of a solution (as detailed in 4.3.1) and feasibility checks on the final states to allow for error detection.

4.1.2. Feasibility

A state s is said to be feasible if the currently packed items for every bin $b \in s.B$ respects the constraints defined in the problem formulation (3.3)

Since the proposed heuristic is constructive it is more convenient to define the concept of feasibility relative to a change in the state.

Insertions Given a state s and $b \in s.B$, an insertion of items is a set of items that are placed in b and have their z_i within a tolerance of a certain z.

Definition 4.4 (Insertion). *Given a state s and a tolerance β_s we define an insertion or placement p a tuple (b, I) where b is a bin and I is a set of items that are going to be packed in b such that, $I \subseteq s.U \land \exists z(z \in \mathbb{Z} \land \forall i(i \in I \land |z_i - z| \leq \beta_s))$*

Observation 4.2. *Given s and $p = (b, \emptyset)$ where $b \notin s.B$, p is an insertion which will open bin b in s.*

Definition 4.5 (Next). *Let p be an insertion over a state s we can then define $s' = Next(s, p)$ as the "copy" of state s with $s'.p = p$. And p is then pending on s'.*

In this way, we can evaluate the changes to the score of a state based on its pending insertion without having to update all the structures for every evaluated state. This property will become apparent in section 4.2.

We can then define an algorithm that applies insertions to a given state s with pending insertions with the help of a function $OpenBin(b)$ which initializes a new structure q_b with every element at its empty value. The proposed algorithm is shown in 1.

Algorithm 1: Commit

---

input : s
output: s'
$(b, I) \leftarrow s.p$
$s' \leftarrow Clone(s)$ `//Memory clone of` s
if $b \in s'.B$ **then**
  $q_b \leftarrow (q_i \in s'.Q : i = b)$
  $q_b.J \leftarrow q_b.J \cup I$
  $s'.U \leftarrow s'.U \setminus I$
end
else Open a new bin
  $s'.B \leftarrow s'.B \cup b$
  $s'.Q \leftarrow s'.Q \cup OpenBin(b)$
end
$s'.p \leftarrow$ none
return s'

Insertion feasibility Describe insertion feasibility givine the sets defined

---

Algorithm 2: Is Insertion Feasible

---

input : $b, I, z, I_{support}, I_{upper}$
output: $isFeasible$
return $true$

State feasibility Describe how to compute the sets efficiently to use the insertion feasibility logic

---

Algorithm 3: Is State Feasible

---

input : s
output: $isFeasible$
return $true$

Proposition 4.1. *A state s' derived by committing a feasible insertion p to a feasible state s is feasible.*

Observation 4.3. *We can always define the empty state s_e where*

$$\begin{cases} s_e.U = I \\ s_e.Q = \emptyset \\ s_e.B = \emptyset \end{cases}$$

and it is always feasible

4.2. Beam Search

Beam Search (BS) is a heuristic tree search algorithm designed for systems with limited memory where expanding every possible node is unfeasible. The idea behind BS is to conduct an iterative truncated breadth-first search where, at each iteration, only a limited number of k nodes is expanded. After the expansion, every new node needs to be evaluated and sorted in order to prune the number of nodes down to the k best ones. The algorithm keeps exploring until no further node can be expanded.

To perform BS one must define the node structure, an expansion function to generate new nodes from existing ones, a ranking between nodes, and a function to determine if a node is final.

A node in the tree can be represented as the state in section 4.1 and eq. (4.1) can be used to determine if a state is final. We also know that a new state s' derived by s by applying a feasible insertion p can be computed as in definition 4.5. This state expansion procedure, with the exception of empty insertions, will generate new states in our tree which will add a positive number of bins or packed items to the solution so, eventually, it will generate a final state.

If the starting state for the search is feasible every new state generated will be feasible and if a final state is found it will be feasible (proposition 4.1). We also note that starting from state s the time complexity to compute feasible insertions can be lower than the complexity required to update the structures that will be used for further expansions (AABB Tree insertion and balancing, memory cloning, etc.) so we modified the standard BS algorithm to separate the expansion phase from the commit phase.

Given S^0 the set of initial states and k the number of best states to expand at each iteration, the described procedure is represented by algorithm 4. As observed in observation 4.3 it's possible to start the search from $S^0 = \{s_e\}$.

Algorithm 4: Beam search

input : S^0, k
output: s_{best}
$S^t \leftarrow S^0$
$S_{final} \leftarrow \emptyset$
repeat
    $S^{t+1} \leftarrow Expand(S^t)$ (algo. 5)
    $S_{final} \leftarrow S_{final} \cup \{s \in S^{t+1} : IsFinal(s)\}$ (def. 4.2)
    $S^{t+1} \leftarrow S^{t+1} \setminus S_{final}$
    $S^{t+1} \leftarrow Sort(S^{t+1})$ (sec. 4.2.1)
    $S^t \leftarrow \emptyset$
    $i \leftarrow 0$
    forall $s \in S^{t+1}$ **do**
        $S^t \leftarrow S^t \cup Commit(s)$ (algo. 1)
        $i \leftarrow i + 1$
        if $i > k$ **then**
            break
        end
    end
until $S^t \neq \emptyset$
$S_{final} \leftarrow Sort(S_{final})$
return *first element of* S_{final}

State Expansion An expansion of a state s can be seen as a new set of states S_{new} derived by a set of feasible insertions. In order to determine these insertions, an underlying heuristic is used (described in section 4.3).

The main idea in this phase of the algorithm is to find feasible insertions in all the bins for items that still need to be packed and that are of the same height. With this approach, the solutions given by the algorithm will start by trying to fill lower layers with items of the same height if possible and they'll become more heterogeneous in upper layers where the classes of height will start to mix up. The underlying heuristic will also use a scoring mechanism to select the best insertions for a given class of heights in order to avoid having too many states to sort.

Given a set of items I and a tolerance β_s we can introduce an algorithm to group them by height and produce a set G of tuples (h, I') where h is the height of the group and I' is the set of items grouped as in algo. 6.

Once items are grouped by height the best insertion for each class of items can be computed for each open bin. If no insertion is possible in any bin, then the only viable insertion is the bin opening insertion (observation 4.2). The described procedure is detailed in algo. 5.

Algorithm 5: Expand

```
input  : S
output: S_new
forall s ∈ S do
    S_new ← ∅
    G ← GroupByHeight(s.U) (algo. 6)
    placed ← false
    forall (h, I) ∈ G do
        forall q_b ∈ s.Q do
            P ← SPBestInsertion(q_b, I) (algo. 7)
            if P ≠ ∅ then
                placed ← true
                forall p ∈ P do
                    S_new ← S_new ∪ Next(s, p) (def. 4.5)
                end
            end
        end
    end
    if placed = false then
        Open a new bin with index |s.B| (oss.  4.2)
        S_new ← S_new ∪ Next(s, (|s.B|, ∅))
    end
end
return S_new
```

Algorithm 6: Group By Height

```
input  : I, β_s
output: G
G ← ∅
forall i ∈ I do
    generate ← true
    forall (h, I′) ∈ G do
        if |h_i − h| ≤ β_s then
            generate ← false
            I′ ← I′ ∪ i
            break
        end
    end
    if generate = true then
        G ← G ∪ (h_i, {i})
    end
end
return G
```

4.2.1. Scoring States

In order to sort states, a scoring function needs to be defined over them. Since the scoring of the states is what will influence the final solution the most, parameters that are directly related to minimizing the objective function are selected.

In the proposed solution to handle multiple objective functions, lexicographic ordering is used.

Definition 4.6. *Let $f_1(s), f_2(s), f_i(s), \ldots, f_n(s)$ be objective functions ordered by precedence based on index i, then*

$$s < s' \textbf{\textit{ iff }} \exists j \in \mathbb{Z} : \begin{cases} f_j(s) < f_j(s') \\ f_k(s) = f_k(s'), \quad \forall k \in \mathbb{Z} : 0 \leq k < j \end{cases}$$

Scoring metrics for each state s that we want to evaluate can then be computed in the *Next* algorithm by considering the contents of the pending insertions and updating each parameter differentially.

The defined ordering utilized is the following:

- $f_1(s) = -|s.B|$: we prefer states that opened fewer bins.
- $f_2(s) = \text{avgvol}(s)$: we prefer states that have packed more average volume between bins.
- $f_3(s) = \text{avgcageratio}(s)$: we prefer states that have better average cage ratio (def. **??**) between bins.

4.3. Support Planes

We introduce Support Planes (SP) which is a heuristic introduced in this thesis based on an underlying 2DBPP heuristic which is used to evaluate feasible expansions of a given node in the BS. The proposed heuristic ensures that the constraint of support isn't violated. The idea at the base of SP is to build a solution to the 3DSPP by filling 2D planes called support planes.

Each support plane can be characterized by the triple $S_z = (z, I_{support}, I_{upper})$ where

- z: the height of the plane
- $I_{support}$: the set of the items that can offer support to items placed on the plane
- I_{upper}: the set of items that will be obstacles to potential new items placed on the plane

Let *coords* be the set of possible coordinate changes which allow for the problem to evaluate placements starting from different corners of the bin.

Given a function $IsFeasible(i, bin, I_{support}, I_{upper}, aabb)$ which evaluates if a packing of item i in bin *bin* is feasible, and the function $ComparePacking(p, p')$ which defines a ranking over placements in the same plane, the SP algorithm can be written as algorithm 7.

Algorithm 7: SP Best Insertion

```
input  : s_b, I
output: placement
placement ← ∅
forall S_z ∈ planes do
    I_p ← I \ {i ∈ I : z + i.h > H_b}
    forall change ∈ coords do
        I'_upper ← CoordinateChange(change, I_upper)
        I'_p ← CoordinateChange(change, I_p)
        P' ← SPPackPlane(W_b, D_b, I'_upper, I'_p) (Algorithm 8)
        P ← CoordinateChange(change, P')
        P ← {i ∈ P : IsFeasible(i, bin, I_support, I_upper, aabb)}
        if ComparePacking(placement, P) then
            placement ← P
        end
    end
    if placement ≠ ∅ then
        return placement
    end
end
return placement
```

To evaluate a packing on a plane a heuristic to solve the 2DBPP is used with the introduction of fixed placements which represent items on other planes that will be obstacles in the current one.

Given the dimensions of the 2D bin (W_b, D_b), the set of obstacles I_o and the set of items to pack I_p a new placement can be computed following algorithm 8

Algorithm 8: SP Pack Plane

input : W_b, D_b, I_o, I_p
output: P
$P \leftarrow \emptyset$
$2dPacking \leftarrow \emptyset$
foreach $i \in I_o$ **do**
    `//Initialize the 2D bin packing instance with each obstable already placed`
    $2DPlaceRect(2dPacking, i)$
end
repeat
    `//Pack untill full`
    $p \leftarrow 2DPackRect(2dPacking, W_b, D_b, i)$
    $P \leftarrow P \cup \{p\}$
until $p \neq \emptyset$
return P

Commit Extension We now describe an extension to *Commit* (algo. 1) to update the structures needed by SP.

When a plane is filled, new insertions become less likely to be feasible. To avoid evaluating planes where no insertion is possible a mechanism to prune dead planes can be introduced.

Since best insertions for a bin are always evaluated by considering lower planes first, if all the insertions in *Expand* (algo. 5) happened over a z_{min} then we can safely remove the opened planes with $z < z_{min}$ for that bin. Let us introduce a z_{min} variable carried over in q_b for each bin, which is updated during the *Expand* phase with the minimum z of all the insertions on bin b. Once the best states are computed and *Commit* is called we can then use its value to prune planes in each q_b. Other operations are also necessary in the *Commit* algorithm to allow SP to update its data structures accordingly to the insertion.

Given a state s and an insertion p where each packed item $i \in p.I$ in bin b has z_i within tolerance of z and the minimum height for the considered bin $q_b.z_{min}$. The algorithm which updates the structures for a given bin b is represented by algorithm 9. This new algorithm can be used as the last step of the *Commit* algorithm for each $b \in s'.B$.

Algorithm 9: SP Apply and Filter

input : $s, p, z, z_{min}, \beta_s$
output: s

$q_b \leftarrow (q_i \in s.Q : i = p.b)$
`//Filter bad planes`
$q_b.Z \leftarrow q_b.Z \setminus \{(z', I_{support}, I_{upper}) \in q_b.Z : z' < z_{min}\}$
`//Apply insertion`
forall $i \in p.I$ **do**
    $q_b.T \leftarrow InsertAABB(i, q_b.T)$ `//If balanced` $O(log(n))$
    $generate \leftarrow true$
    forall $(z', I_{support}, I_{upper}) \in q_b.Z$ **do**
        `//Based on the distance from the top of the item`
        $dz \leftarrow z' - (z_i + h_i)$
        if $0 \leq dz \leq \beta_s$ **then**
            $generate \leftarrow false$
            $I_{support} \leftarrow I_{support} \cup i$
        end
        else if $dz < 0$ **then**
            $I_{upper} \leftarrow I_{upper} \cup i$
        end
    end
    if $generate$ **then**
        $q_b.Z \leftarrow q_b.Z \cup (z_i + h_i, \{i\}, \emptyset)$
    end
end
return s

4.3.1. Scoring Insertions

5 | Computational experiments

6 | Conclusions and future developments

A final chapter containing the main conclusions of your research/study and possible future developments of your work have to be inserted in this chapter.

Bibliography

[1] G. v. d. Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of graphics tools*, 2(4):1–13, 1997.

A | Appendix A

If you need to include an appendix to support the research in your thesis, you can place it at the end of the manuscript. An appendix contains supplementary material (figures, tables, data, codes, mathematical proofs, surveys, . . .) which supplement the main results contained in the previous chapters.

B | Appendix B

It may be necessary to include another appendix to better organize the presentation of supplementary material.

List of Figures

List of Tables

List of Symbols

Variable	**Description**	**SI unit**
$\boldsymbol{u}$	solid displacement	m
$\boldsymbol{u}_f$	fluid displacement	m

Acknowledgements

Here you might want to acknowledge someone.