A Two-Stage 3D Bin Packing Algorithm

for Groceries at Online Supermarket Picnic

Master Thesis Computer Science

Daan Goslinga

A Two-Stage 3D Bin Packing Algorithm
for Groceries at Online Supermarket Picnic

by

Daan Goslinga

Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
November, 2021 - July, 2022

TU Delft

Preface

Over the past 7.5 months I have been working on this thesis as an intern at Picnic. During my time at this company I have learned a lot in terms of software development and project management. The lessons learned and guidance I have gotten would not have come to me if it wasn't for a group of people who supported me along the way. This preface serves as a token of gratitude to these individuals.

First of all, I would like to thank Neil-Yorke Smith who was my supervisor from TU Delft. His guidance and flexibility has really helped me in creating my own thing, while still being in line with the expectations of a thesis. I have found that he not only cared about the thesis itself, but always found my personal well being equally important. This has been greatly appreciated and made me truly enjoy our collaboration.

Secondly, I would like to thank Picnic and more specifically the DIST team that was so kind to adopt me as an intern. I had great time spending time with you, from the daily standups to late après-ski parties. I am especially grateful to Nadir Belarouci, who at a later stage of my thesis was willing to take over the supervisor role. The fact that this switch in supervision happened without any hiccups is a testament to how committed he has been at guiding me from the start. Furthermore, I would like to thank him for the chess games we played which humbled me greatly.

Lastly, I would like to thank my family and friends for their support. I can not wait to share my thesis with you as to show what I have been doing these last few months.

Daan Goslinga
Delft, June 2022

Abstract

The online supermarket Picnic uses the volume of groceries to pack them into totes and bags. While this generally works well, there are cases in which an article's volume fits into a designated space, but its dimensions do not. To prevent this so-called overflow, Picnic uses a maximum fill rate of 85%, which means that no tote can be filled to more than 85% of its volume. Unfortunately, the use of a maximum fill rate does not guarantee that no overflow occurs at all and workers are sometimes still instructed to perform infeasible packings. To solve this, this thesis proposes a Three-dimensional Bin Packing algorithm (3D-BAGS) which uses the width, height and length of an article to determine whether it can be placed inside a bag. A Biased Random-Key Genetic Algorithm is used to converge to a good packing solution. After which, the same algorithm divides the bags into so-called totes, which are shipped to the customer. 3D-BAGS expands upon the state-of-the-art Three-dimensional Bin Packing algorithms by including bag stretching plus the diagonal rotation and squeezing of articles. Computation time of the current state-of-the-art is reduced by including memoization, priority based multithreading and a different stopping criterion. This thesis shows that the use of 3D-BAGS leads to more accurate packings, while simultaneously lowering the amount of totes needed to ship ambient and chilled groceries. However, 3D-BAGS needs more totes to ship frozen products compared to Picnic's current approach and is also more computationally expensive.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
BRKGA	Biased Random-Key Genetic Algorithm
DFTRC	Distance to the Front-Top-Right Corner
EMS	Empty Maximum Space
EPV	Electric Picnic Vehicle
MOTPE	Multi Objective Tree-structured adaptive Parzen Estimators
TPE	Tree-structured adaptive Parzen Estimators
VND	Variable Neighborhood Descent

List of Figures

List of Tables

1 Introduction

Ever since the introduction of Amazon and eBay in 1995, online shopping has shown an increasing trend in popularity worldwide. This trend has only been accelerated more by the lockdowns imposed because of the COVID-19 pandemic. While in the Netherlands online shopping started out to be mainly focused on non-food related consumption, in the recent years supermarkets have seen their share of online orders rise significantly. One of the biggest online supermarkets currently on the Dutch market is Picnic. A unique characteristic of this supermarket is that it is exclusively online, something that as of recently can be seen a lot more often in the so called 'flitsbezorgers'. While these 'flitsbezorgers' serve a vastly different need than Picnic, namely to deliver small amount of groceries in a short time span, they do show that the idea of exclusively online supermarkets is likely here to stay.

In order to deliver groceries to its customers, Picnic packs deliveries into crates which are stacked into frames that are shipped to the customers. To divide deliveries over these crates, a so called bin packing algorithm is used. The focus of this thesis lies within this bin packing algorithm. While Picnic's current algorithm packs groceries based on their volume, this thesis proposes an algorithm that packs articles based on their width, height and length.

Before diving into existing literature and the algorithm that is created as a product of this thesis, an introduction is given on how Picnic operates and a more detailed explanation is given of its current approach to bin packing. This section serves to provide the necessary context as to understand why this thesis is needed from Picnic's perspective, while Chapter 2 motivates its scientific need.

The following three parts are discussed in this introduction. First, some background information is given on some of the relevant processes used by Picnic. After this, a summary of their current bin packing algorithm is given. Finally, the problems that arise from the use of this algorithm are analysed and the goal of this thesis is outlined.

1.1. Background

1.1.1. Totes

The crates that Picnic uses to store and move groceries are called 'totes'. Each tote contains three bags, which are generally $1/3$ the length of a tote but can stretch to $1/2$ of a tote. A tote is assigned to one of three temperature zones: ambient, chilled or frozen. While ambient totes are always open on the top, chilled and frozen totes need to be closeable with a lid for insulation purposes. A tote can only contain articles that have the same temperature zone. A bag has the same width and height as a tote of a similar temperature zone, but its length is $\frac{1}{3}$ of said tote.

The exact dimensions of a tote per temperature zone are provided in the confidential version of this thesis.

1.1.2. Trips and deliveries

Multiple totes are grouped together in a delivery, which is the order of a single customer. These deliveries are then grouped together into trips, which are routes an Electric Picnic Vehicle (EPV) traverses in order to bring the groceries to different customers.

1.1.3. Multi-delivery totes

Originally, Picnic would assign at most one customer to a tote. This would lead to instances in which an entire tote would only be partly used because orders were not big enough to fill all three bags. To make more efficient use of totes, Picnic introduced multi-delivery totes. These totes contain bags of different customers. By merging totes of different customers into one tote, Picnic reduces the amount of unused space and thus the amount of necessary totes. A visual example of this can be found in Figure 1.1.

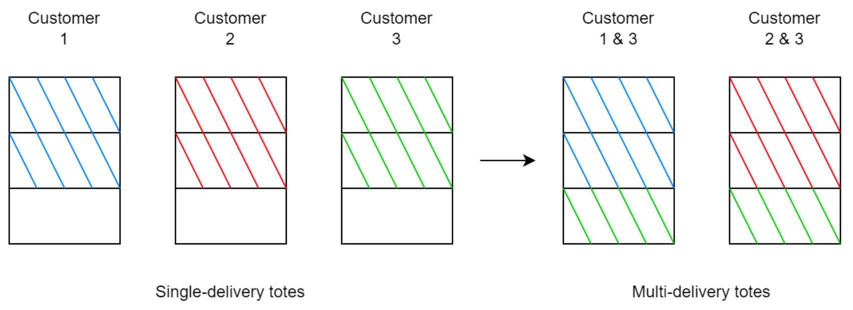


Figure 1.1: A visual example of multi vs. single-delivery totes.

While different orders can be combined into one tote, different temperature zones can not. This means that every tote can only contain bags of the same temperature zone.

1.1.4. Picking zone (fragility)

At a Picnic fulfillment centre, articles are stored in so-called 'picking zones'. Picking zones are sorted by volume, fragility and weight. When packing articles into bags, a picker starts at the picking zone containing the biggest, least fragile and heaviest items and ends at the picking zone with the smallest, most fragile and lightest items. Because of this, the heaviest articles will be on the bottom of a bag while the most fragile/lightest articles will be placed on top. Figure 1.2 gives a top down example of picking zones and the route a picker traverses.

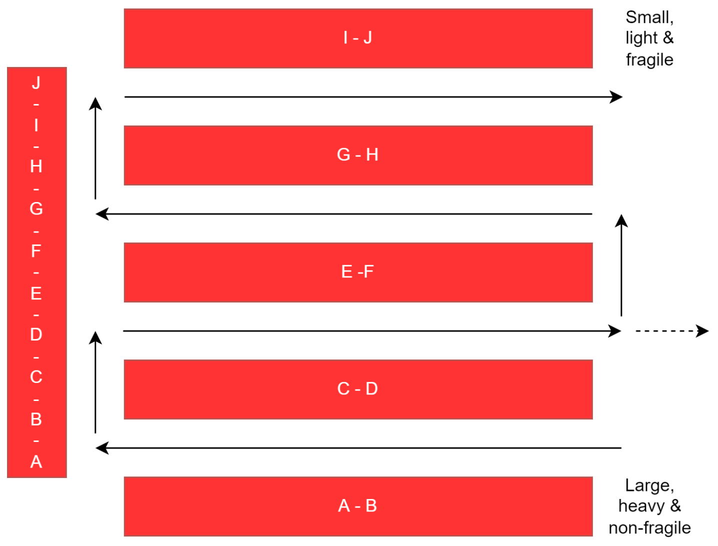

I - J
Small, light & fragile
G - H
E -F
C - D
A - B
Large, heavy & non-fragile
J - I - H - G - F - E - D - C - B - A

Figure 1.2: Top down view of picking zones and the route a picker traverses.

Pickers who have packed all articles before reaching the final picking zone can exit the path earlier at an exit point. This prevents pickers from having to visit all picking zones while not needing to pick any more articles.

1.2. Current approach

Currently, Picnic uses a bin packing algorithm that only accounts for the volume and weight of each article and tote. The goal is to divide the articles of a delivery in as few totes as possible, while preventing a tote from being too full or heavy.

A detailed description of Picnic's current approach is provided in the confidential version of this thesis.

1.3. Motivating problems

While Picnic's current strategy can lead to good packings in an acceptable time frame, some solutions can lead to problems at an operational level (i.e., packing the totes in real life). A list of these problems is given below.

1.3.1. Overflow

The current algorithm used by Picnic only accounts for an article's volume when it determines whether an article can fit in a bag. Because an article's dimensions are not considered, there are instances in which an article's volume can fit but its dimensions can not.

On my first week at Picnic, I worked at a fulfillment centre for a day to get a feel for the packing process. During this day I also encountered overflow while packing a baguette. While its volume is relatively small, its length is quite big. The bag to which the baguette was assigned was already somewhat full but could still include the volume of the baguette. However, because of its length the baguette actually stuck out of the bag more than it was allowed. Since totes are stacked on frames, it was not an acceptable packing and I was forced to reshuffle articles. Other employees have also encountered the same kind of problems with different articles. Two more examples are given below.

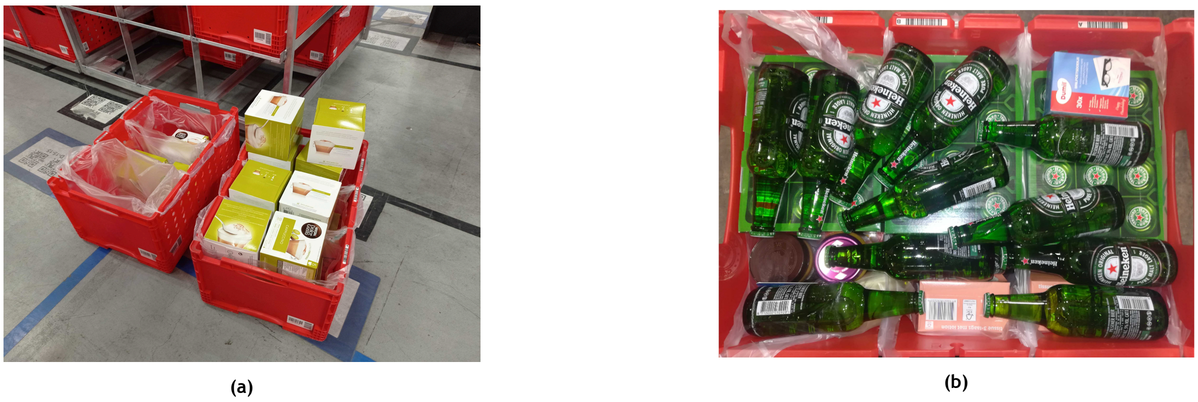


Figure 1.3: Two examples of overflow given cappuccino cups and a beer case.

In Figure 1.3a fourteen boxes of cappuccino cups were assigned to one tote. As can be seen, roughly four boxes can fit in one bag with some space left at the right. Because of the shape of the boxes, two of the fourteen boxes could not fit in the tote. In Figure 1.3b a case of beer was assigned to a bag. However, due to underlying items, the case stuck out more than it was allowed to. In order to have it fit, the case was opened and the beer was distributed over the whole tote. While opening the case solves the infeasible packing, it costs extra time and is actually against Picnic's policy to do so.

In all three cases, the need for reshuffling costs extra time and thus money. Further, the need for an extra tote could lead to orders not fitting in an Electric Picnic Vehicle and thus infeasible trips.

Fill rate

In order to prevent overflow as much as possible, Picnic imposes a maximum fill rate of 85%. This means that a tote can only be filled up to 85% of its volume. The idea is that if overflow occurs, there is still some volume of a tote left in which these articles that cause overflow can be placed without any problems. While this generally has been working well for the company, there are two disadvantages to this approach. Firstly, as can be seen in the examples above, a maximum fill rate does not guarantee that no overflow can occur at all. Secondly, by imposing such a maximum fill rate, all totes shipped by Picnic contain at least 15% of unused space. From a business perspective, it would be better to find a way to raise the maximum fill rate such that that this unused space could also be utilized. By allowing close to 100% of a tote to be used for packing, one is likely to need less totes and thus less EPVs to ship the groceries to customers, which leads to a decrease in costs.

1.3.2. Oversized articles

Picnic sells articles that are bigger in one or more dimensions than a single bag. Because Picnic's current algorithm only accounts for volume, it does not know that these oversized articles can actually not fit into a bag. Most of the time this is not a problem, since articles can be squeezed to fit into a bag. However, not all oversized article (e.g., baguettes) are squeezable. Operationally this is often solved by having the bag stretch to more than $\frac{1}{3}$ of a tote, but sometimes this is not possible due to other bags also needing that space. These unsqueezable oversized articles can lead to overflow even if there is a lot of unused volume left.

1.3.3. Damaged articles

To prevent overflow, pickers might squeeze articles that are actually not squeezable. In my own experience, I noticed quite a few frozen products needed to be squeezed in order to fit into a bag. This is of course undesirable because Picnic wants to deliver its articles in the most pristine state as possible to ensure customer satisfaction.

1.3.4. Inefficient packings

Because the current algorithm used by Picnic only accounts for volume (and weight), it does not know where to exactly place an article during the packing process. Pickers are only told in which bag they should pack an article and not where an article should be placed in said bag. This can cause some inefficiencies due to human error. While Picnic is not planning on instructing pickers where to exactly place articles inside a bag, it could be necessary to know exact placements once picking is done by robots instead of humans. This is not possible with its current algorithm.

1.4. Goal

The goal of this thesis is to create a packing algorithm that represents a more accurate model of reality. By taking the width, height and length of articles into account when generating packings, one can be sure that there exists a packing in which no overflow occurs. Generating packings based on an article's dimensions is called Three-dimensional Bin Packing. Using this technique would not only lead to more accurate packings, it would also allow Picnic to raise their maximum fill rate, which could potentially lead to less totes being needed to ship the same amount of groceries. Also, by knowing for sure whether an article can fit without squeezing, it is less likely to be damaged during packing. Lastly, Three-dimensional Bin Packing could allow Picnic to instruct their pickers (or even robotic workers) where to exactly place articles in a bag, if the company desires to do so in the future.

Furthermore, another goal of this thesis is to find a way to pack unsqueezable articles that have one or more dimensions that are bigger than the bag in which they need to be packed. Developing methods that can be used to pack such oversized articles leads to a more accurate model of reality because the amount of unsqueezable oversized articles that are squeezed to fit in a bag is reduced.

The algorithm that is developed for this thesis should ideally not take longer than 1 minute to process 13 trips, as is the current maximum time allocation for Picnic's use case. However, this time limit is not a hard constraint and could be raised if needed.

The main research question this thesis seeks to answer is:

1. *What Three-dimensional Bin Packing algorithm could be used to efficiently pack groceries into totes and how should this algorithm be expanded upon to fit the Picnic use case?*

The sub-questions of this thesis are:

2. *Can Three-dimensional Bin Packing lead to a fill rate higher than 85% given Picnic's product range?*
3. *Does Three-dimensional Bin Packing generate packings with less totes compared to Picnic's current approach?*
4. *How can a Three-dimensional Bin Packing deal with articles that have one or more dimensions that are bigger than the bag in which they need to be placed*?

These questions are answered by first conducting a literature review, which is done in Chapter 2. Based on this review, a scientific gap is found which needs to be filled in order to obtain the answers to the given questions. Chapter 3 explains how this thesis fills this gap by expanding upon state-of-the-art algorithms found in the literature review. This new algorithm is called 3D-BAGS. 3D-BAGS requires hyperparameter optimization to reach its full potential. This optimization is discussed in Chapter 4. Once the best set of hyperparameters for the packing algorithm is found, the experimental results are given in Chapter 5. The research questions are then answered, based on these results, in Chapter 6. Appendix A contains a side study in which 3D-BAGS is used to separate contaminated articles from food. However, due to confidentiality this study is only published in the confidential version of the thesis. Furthermore, Appendix B gives an outline of a scholarly paper that could be published as a result of this thesis.

2 Related work

2.1. Problem variants

Three-dimensional Bin Packing, also known as the Multi-Container Loading Problem, has numerous real world applications and is thus very well researched in the scientific community. Some examples of this are the stacking of air cargo [34], the loading of pallets [21] and the stocking of transport trucks [24]. There are also many variants to this problem in either 1D or 2D. However, since the focus of this thesis is on Three-dimensional Bin Packing, only 3D variants are covered. The most common and relevant of those are discussed below:

1. ***Three-dimensional Bin Packing***, in which items have to be packed into bins, according to their width, height and length. The goal is to minimize the amount of used bins. Bin sizes can either be the same or different per bin. In almost all literature item size is heterogeneous.
2. ***Three-dimensional Knapsack Packing***, which tries to pack a single bin as efficiently as possible. It is also known as the Single Container Loading Problem. This variant can easily be converted to Bin Packing by opening a new bin if no items fit in the current bin anymore. Vice versa, Bin Packing can also be converted to Knapsack Packing by limiting the amount of bins to one.
3. ***Three-dimensional Strip Packing***, also known as the Three-dimensional Open Dimension Problem, assumes two dimensions of a bin are fixed and one dimension is infinite. The goal is to minimize the packing size of the infinite dimension.

These variants can be divided into two groups: offline and online algorithms. An offline algorithm knows at the beginning of its execution what the exact input data is. Online algorithms receive their input in parts during execution. More concretely, for Three-Dimensional Bin Packing, offline algorithms know the exact list of items that need to be placed and in which order this must be done. Online algorithms, obtain one (or a few) item(s) at a time and places them the moment they are received. Offline algorithms perform generally better than online algorithms since the knowledge of the items and their order can be exploited to find the most optimal solution. For Picnic's use case, the articles and their placement order are known beforehand. Thus, this literature review only discusses offline algorithms.

2.2. Formal Problem Description

A formal description of Three-dimensional Bin Packing has been described by Chen et al. [10]. This formal description is a mixed integer programming model and has been expanded upon by Mahvash et al. [30], who included rotation of items. Tables 2.1 and 2.2 state the used input and decision variables respectively. After which the objective function and constraints are provided.

Table 2.1: The input variables of the mixed integer programming model.

Input variables	Description	Type	Range
I	The set of items i.	$\mathbb{Z}^+$	$[0,\infty)$
w_i, h_i, l_i	The width, height and length of item i.	$\mathbb{Z}^+$	$[0,\infty)$
W, H, L	The width, height and length of the bins.	$\mathbb{Z}^+$	$[0,\infty)$

Table 2.2: The decision variables of the mixed integer programming model.

Decision variables	Description	Type	Range
x_i, y_i, z_i	The origin coordinate of item i, representing its Back-Lower-Left corner.	$\mathbb{Z}^+$	$[0,\infty)$
n_i	The bin in which item i is packed.	$\mathbb{Z}^+$	$[0,\infty)$
nB	The amount of bins used.	$\mathbb{Z}^+$	$[0,\infty)$
M	Any sufficiently large number.	$\mathbb{Z}^+$	$[0,\infty)$
o_{i1}, o_{i2}, o_{i3}, o_{i4}, o_{i5}, o_{i6}	The rotation of item i. Only one of these variables can be one, while the rest must be zero. An overview of the rotations can be seen in Figure 2.1.	Binary	$[0,1]$
$f^1_{ii'}$, $f^2_{ii'}$, $f^3_{ii'}$, $f^4_{ii'}$, $f^5_{ii'}$, $f^6_{ii'}$	The position of item i relative to item i'. These variables dictate whether item i is behind, in front, to the left, to the right, below or above item i'.	Binary	$[0,1]$
$s_{ii'}$	Whether item i is in the same bin as item i'.	Binary	$[0,1]$
δ	A binary variable used to make sure that $s_{ii'}$ is only 1 if two items are in the same bin.	Binary	$[0,1]$

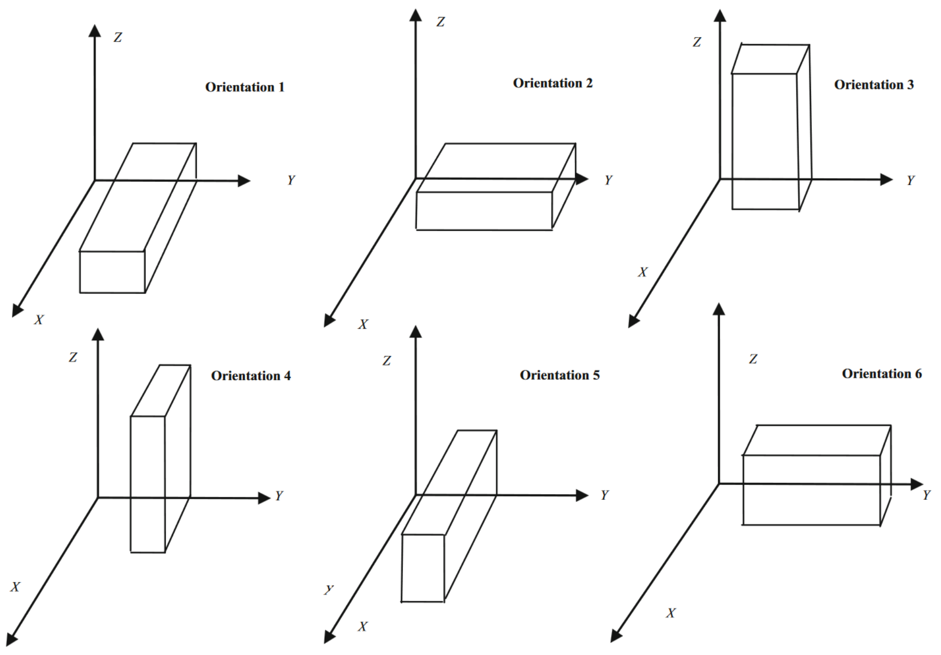


Figure 2.1: The six possible orientations of an item (from Mahvash et al. [30]).

The objective function is as follows:

$$Minimize \quad nB \tag{2.1}$$

The objective function minimizes the number of bins (nB) needed to pack all items. The number of bins is defined by the amount of unique bin labels (n_i).

Subject to the constraints:

$$x_i + l_i(o_{i1} + o_{i5}) + w_i(o_{i2} + o_{i4}) + h_i(o_{i3} + o_{i6}) \leq L \qquad \forall i \in I \tag{2.2}$$

$$y_i + l_i(o_{i2} + o_{i6}) + w_i(o_{i1} + o_{i3}) + h_i(o_{i4} + o_{i5}) \leq W \qquad \forall i \in I \tag{2.3}$$

$$z_i + l_i(o_{i3} + o_{i4}) + w_i(o_{i5} + o_{i6}) + h_i(o_{i1} + o_{i2}) \leq H \qquad \forall i \in I \tag{2.4}$$

These three constraints ensure that every placed item exists within the bounds of a bin. It does this by stating that the start location of an item plus its width, height or length, given its rotation, should never exceed the maximum size of each dimension.

$$x_i + l_i(o_{i1} + o_{i5}) + w_i(o_{i2} + o_{i4}) + h_i(o_{i3} + o_{i6}) \leq x_{i'} + (1 - f^1_{ii'})M \qquad \forall i, i' \in I, i < i' \tag{2.5}$$

$$x_{i'} + l_{i'}(o_{i'1} + o_{i'5}) + w_{i'}(o_{i'2} + o_{i'4}) + h_{i'}(o_{i'3} + o_{i'6}) \leq x_i + (1 - f^2_{ii'})M \qquad \forall i, i' \in I, i < i' \tag{2.6}$$

$$y_i + l_i(o_{i2} + o_{i6}) + w_i(o_{i1} + o_{i3}) + h_i(o_{i4} + o_{i5}) \leq y'_i + (1 - f^3_{ii'})M \qquad \forall i, i' \in I, i < i' \tag{2.7}$$

$$y_{i'} + l_{i'}(o_{i'2} + o_{i'6}) + w_i(o_{i'1} + o_{i'3}) + h_{i'}(o_{i'4} + o_{i'5}) \leq y_i + (1 - f^4_{ii'})M \qquad \forall i, i' \in I, i < i' \tag{2.8}$$

$$z_i + l_i(o_{i3} + o_{i4}) + w_i(o_{i5} + o_{i6}) + h_i(o_{i1} + o_{i2}) \leq z_{i'} + (1 - f^5_{ii'})M \qquad \forall i, i' \in I, i < i' \tag{2.9}$$

$$z_{i'} + l_{i'}(o_{i'3} + o_{i'4}) + w_{i'}(o_{i'5} + o_{i'6}) + h_{i'}(o_{i'1} + o_{i'2}) \leq z_i + (1 - f^6_{ii'})M \qquad \forall i, i' \in I, i < i' \tag{2.10}$$

These constraints prevent placed items from overlapping with each other. To do so, it must be that the item that is behind another item, in any dimension, can not have an end coordinate that is bigger than the start coordinate of the item it is behind.

$$f^1_{ii'} + f^2_{ii'} + f^3_{ii'} + f^4_{ii'} + f^5_{ii'} + f^6_{ii'} \geq s_{ii'} \qquad \forall i, i' \in I, i < i' \tag{2.11}$$

This constraint ensures that overlap of items is only checked when they are in the same bin.

$$o_{i1} + o_{i2} + o_{i3} + o_{i4} + o_{i5} + o_{i6} = 1 \qquad \forall i \in I \tag{2.12}$$

This constraint allows items to be placed with only one rotation.

$$1 \leq n_i \leq nB \qquad \forall i \in I \tag{2.13}$$

This constraint ensures that the label of a bin in which an item is placed is higher or equal to one and lower or equal than the number of bins.

$$n_i - n_{i'} \leq (1 - s_{ii'})M \qquad \forall i, i' \in I, i < i' \tag{2.14}$$

$$n_{i'} - n_i \leq (1 - s_{ii'})M \qquad \forall i, i' \in I, i < i' \tag{2.15}$$

$$n_i - n_{i'} < (s_{ii'} + \delta)M \qquad \forall i, i' \in I, i < i' \tag{2.16}$$

$$n_{i'} - n_i < (s_{ii'} + (1 - \delta))M \qquad \forall i, i' \in I, i < i' \tag{2.17}$$

The above constraints make sure that $s_{ii'}$ is one when item i and item i' are placed in the same bin (n_i = $n_{i'}$) and zero if not.

$$\delta, f^1_{ii'}, f^2_{ii'}, f^3_{ii'}, f^4_{ii'}, f^5_{ii'}, f^6_{ii'}, s_{ii'} \in 0, 1 \qquad \forall i, i' \in I \tag{2.18}$$

$$x_i, y_i, z_i, n_i, nB \in \mathbb{Z}^+ \qquad \forall i \in I \tag{2.19}$$

The last two constraints indicate which values the variables are allowed to have.

2.3. Placement locations

In order to choose a placement location, algorithms should know which locations are available. There are multiple methods to determine placement locations. In the following section four commonly used approaches in literature are discussed. The section starts with corner points. After this, an extension to this approach, called extreme points, is explained. The final two discussed approaches are the use of empty maximum spaces and a top down view of a bin given by a matrix.

2.3.1. Corner points

Martello et al. [31] propose a placement algorithm that generates so called ‘corner points’ on which items are allowed to be placed. To find these points it first calculates all the end points of every item that is already packed. For each item i in a 2D environment, its end point is $(x_i + w_i,\ y_i + l_i)$. Where x_i and y_i are the origin location of an item and w_i and l_i are its width and length. Based on the generated end points, an envelope can be observed, which is the area in which no items can be placed anymore.

An example can be found in Figure 2.2, where the slope of the envelope is represented as dotted lines. The envelope has multiple points in which its slope goes from being vertical to horizontal. The items with end points that have the same x or y value as these points, are called extreme items. In this example the extreme items are 1, 3, 4, 6 and 9. After extreme items are found their corner points are calculated, which are the points where lines leading out of the end points of extreme items intersect. These corner points are indicated by black dots in Figure 2.2.

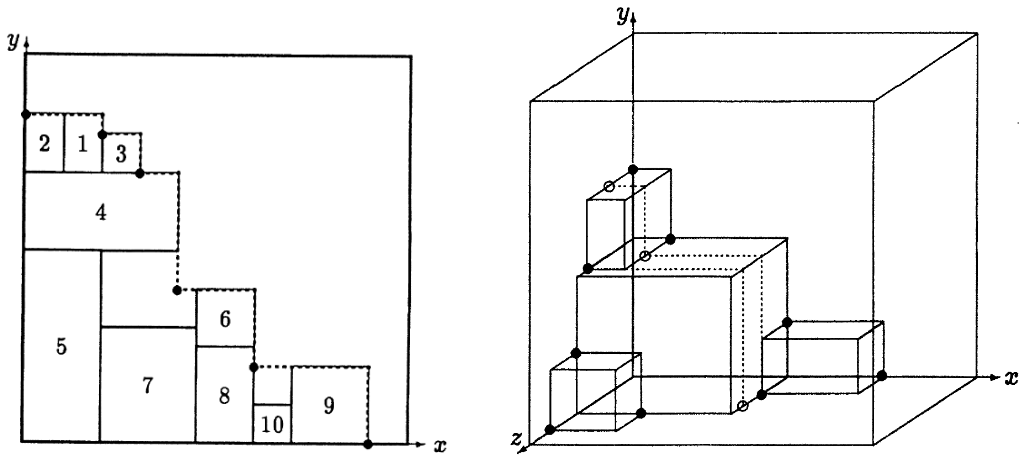


Figure 2.2: 2D and 3D example of corner points (From Martello et al. [31]).

To obtain corner points in a 3D environment, one can run the corner points algorithm for every z coordinate in which an item ends and once at the origin (z = 0). If there are multiple corner points along the same z axis, the corner point with the lowest z value is kept and all other corner points are deleted. This also removes so called false corner points, which are corner points generated by the 2D algorithm which actually do not exist in the 3D environment. In Figure 2.2 the false corner points are indicated by the white dots.

According to Martello et al., finding corner points when placing an item in a 2D environment has a time complexity of $O(n)$, because for every item that is already placed it is determined whether it is extreme and what its end points are. In a 3D environment, the time complexity is $O(n^2)$ when placing an item. The reason is that the 2D algorithm, which is $O(n)$, is ran for every z coordinate in which an item ends, which is also $O(n)$. Since this algorithm is ran for every item that needs to be placed, its time complexity is $O(n^3)$, where n is the amount of articles to be placed.

2.3.2. Extreme points

The concept of extreme points is an expansion of corner points. Crainic et al. [11] realised that when using corner points, some space would be unused even if there was still room for (small) items. An example can be seen in Figure 2.3.

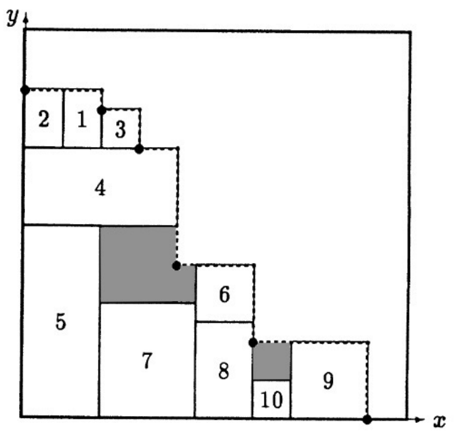


Figure 2.3: Unused space when using corner points.

To circumvent this issue Crainic et al. project extreme points for every newly placed item on the other placed items along the axes of the bin. In a 3D environment, six extreme points are generated per item i:

- Two points generated from $(x_i + w_i,\ y_i,\ z_i)$ coordinate, projected on the y and z axes
- Two points generated from $(x_i,\ y_i + l_i,\ z_i)$ coordinate, projected on the x and z axes
- Two points generated from $(x_i,\ y_i,\ z_i + h_i)$ coordinate, projected on the x and y axes

Where x, y and z are the origin points of the placed item and w, l, h are its width, length and height respectively.

Looking back at Figure 2.2, one could see that after placing item 7 a new extreme point is projected from its top left $(x_i, y_i + l_i)$ corner along the x-axis to the left. This projection immediately hits item 5 and thus generates an extreme point at the top left corner of item 7. Crainic et al. state the the time complexity of placing items according to extreme points is $O(n^3)$, where n is the amount of items.

2.3.3. Empty maximum spaces

Empty maximal spaces (EMS) are the largest possible cuboids in which no item is placed, which was originally proposed by Lai and Chan [27]. While different approaches exist on how to generate EMS [29], the general idea stays the same: The algorithm keeps track of all EMS, which initially is the entire bin. When an item has to be placed, a fitting EMS is chosen as placement location. After the placement, new EMS are generated and the EMS in which the item is placed is deleted. Originally, EMS did not account for vertical stability. Gonçalves and Resende [20] propose generating EMS based on full support. Full support means that the floor of an EMS is fully supported by other items or the floor of the bin. Figure 2.4 gives a visual depiction of newly generated EMS.

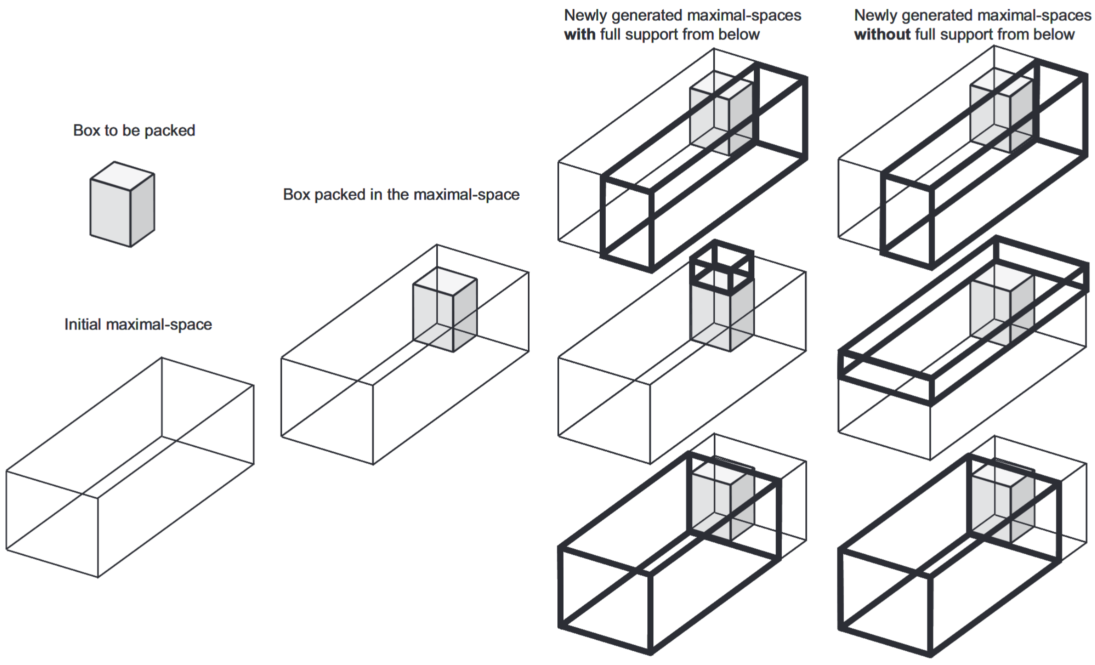

Box to be packed
Initial maximal-space
Box packed in the maximal-space
Newly generated maximal-spaces **with** full support from below
Newly generated maximal-spaces **without** full support from below

Figure 2.4: Visual depiction of empty maximum spaces (From Gonçalves and Resende [20]).

The time complexity for generating EMS is not mentioned by any author. However, one can proof that the complexity for the generation of EMS has a (loose) upper bound of $O(6^n)$, where n represents the amount of items placed. In the worst case, six new EMS are generated when an item is placed inside one EMS, namely one at each side of the item. This can be done in constant time. In this worst case, an item intersects all EMS when placed and thus every EMS is split into six new EMS. Since every time an item is placed the amount of EMS is at most multiplied by six, the generation of EMS can never have a higher time complexity than $O(6^n)$.

This complexity seems much higher than the $O(n^3)$ reported for corner and extreme points. However, in practice, this approach has a much lower time complexity. Most of the time, the algorithm will only create two, three or four new EMS per intersected EMS, since generally three out of six faces of an item are fully covered (as can be seen in Figure 2.4 as well). Also, not every EMS is always intersected when an item is placed. It was found that in practice the amount of EMS roughly grows with a complexity of $O(n^{1.55})$, given a bin with practically infinite dimensions and 200 items of various shapes.

2.3.4. Top down matrix

Another way to find placement selections is by creating a matrix which represents a bin viewed from the top. Verma et al. [38] use this approach to find item placements which are robot-packable in an online environment, but it can also be used in an offline algorithm. Figure 2.5 shows how an item is placed using a top down matrix.

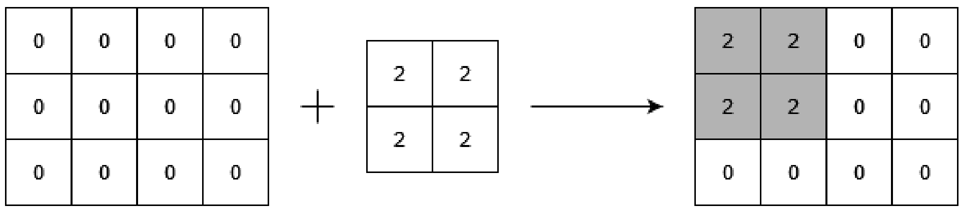


Figure 2.5: Top down view of an item placed inside an empty bin.

In terms of stability, Verma et al. propose to allow placements when all four corners are supported by one or more items, even if the rest of the base is not supported. An example of this can be seen in Figure 2.6.

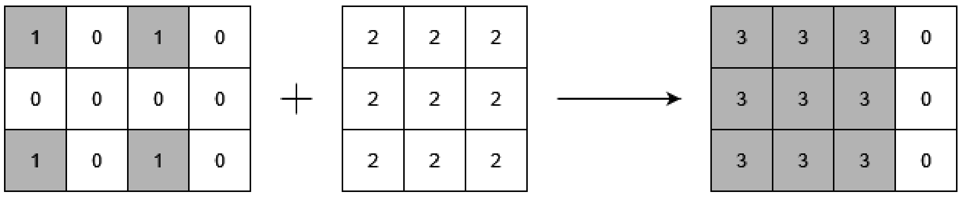


Figure 2.6: Item placement with four supported corners.

While a matrix is an intuitive representation of a top down view, it can lead to inaccuracies when an item does not fit exactly in one or more squares. If an item does not exactly fit, the algorithm has to round its dimensions up so that it corresponds to the squares. Not only will the algorithm assume that more space is used then actually is. It can also assume a corner is fully supported while in reality the supporting items are smaller than the matrix indicates. One could overcome this by making the squares smaller which leads to smaller round-off errors. However, this would also lead to more squares to process and thus requires more computation time. Another issue that can occur when using a top down view is that the model is not aware of any unused space that is below an item, which can lead to possible fitting locations not being found.

No time complexity is given by the authors, but one could reason that in the worst case it has a time complexity of $O(w * h * n)$, where w is the width, h is the height of the matrix and n is the amount of items. In the worst case, all items are one cell big, thus for every item ($O(n)$) it needs to iterate over all cells in a matrix to find a placement location ($O(w * h)$). Combined this leads to a time complexity of $O(w * h * n)$. However, similarly to EMS, this complexity is most likely lower in practice. One could argue that not a single item is one cell big, given that cells are in millimetres. However, if a cell is not square shape, the time complexity worsens again since every rotation should be used to traverse all placement locations in the matrix. One could also define a cell in centimetres instead of millimetres, which means the amount of cells to traverse decreases dramatically. However, this will lead to rounding errors if items are rounded up or down to centimetres.

Figure 2.7 gives a visual overview of the time complexities of each heuristic used to generate placement locations. For the top down matrix, the values for the width and height are equal to the dimensions of an ambient bin. The exact values can be found in the confidential version of this thesis.

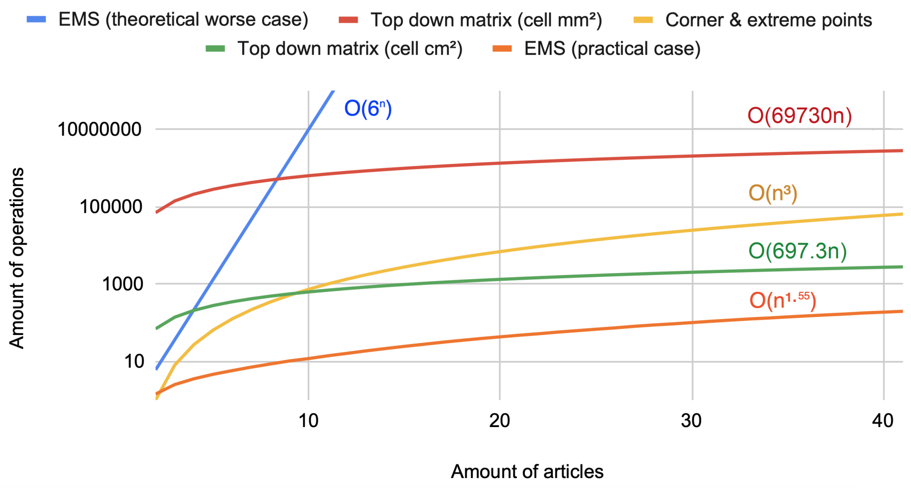


Figure 2.7: A visual overview of the time complexities of each of the placement location heuristics. Note that this graph is in a logarithmic scale.

As can be seen, the practical case of EMS generation has the lowest time complexity for 40 items. For more than 40 items, using a top down matrix, given articles can be the size of one cell and a cell is in cm^2, eventually needs the least amount of operations. However, using this approach can lead to rounding errors and thus inaccuracies. Corner & extreme points have a worse time complexity than the two formerly mentioned placement heuristics but require less operations than the use of a top down matrix in mm^2. In theory, EMS has the worst time complexity in the worst case scenario, however it is unlikely that six EMS are generated per intersected EMS when an item is placed.

2.4. Placement selection

Multiple approaches have been suggested to select which location is chosen for a new item placement. The most common and best achieving strategies are discussed below.

First Fit simply selects the first location in which an item fits. Because of its simplicity, it is easy to implement and requires little computation time, but it can choose placements which lead to unused/wasted space.

Distance to the Front-Top-Right Corner chooses the location furthest away from the Front-Top-Right corner [19]. This strategy leads to items being packed at the Back-Lower-Left corner and evenly spreads out in all axes. Results in terms of utilization rates are very good, but it can lead to uneven weight distribution when not many items are packed.

Layer Building, as the name suggests, tries to fill a bin layer per layer. It starts by filling the floor (which is why it is also called floor building) and once fully covered it builds the next layer on top of it. Layer building is useful when items have somewhat the same size, but if items have different dimensions it quickly leads to layers with uneven surfaces. Because of these uneven surfaces it becomes hard to build new (stable) layers.

Column Building tries to build columns of items which are as high as possible. While in terms of utilization rate, it performs fairly good, it often leads to item placements with very bad stability.

First Fit + Column + Layer Building is also a viable strategy, which combines elements of all three approaches. Verma et al. [38] propose an algorithm which minimizes the differences in heights of items when placed next to each other. This results in columns if item placements are tall but lead to small height differences with surrounding items. Besides this, it penalizes the height of a placed item, so if no smooth surfaces can be created with tall objects, it prefers to do layer building. Lastly, it also penalizes the distance of the placed item to the origin, this leads to some first fit like behaviour.

2.5. Exact algorithms

Exact algorithms are algorithms that are guaranteed to lead to the optimal solution given sufficient time. In terms of Three-dimensional Bin Packing, the most naive approach would be to try out every possible combination of items and select the combination with the lowest bin count. However, because every single combination has to be tried out, this kind of a naive approach is often too slow to have any practical use for big solution spaces. Still, looking at exact algorithms can be useful to get a better understanding of a certain problem and can also give some indication whether any inexact algorithm is performing relatively well.

2.5.1. ONEBIN + Main tree branching

The ONEBIN algorithm is an exact branch-and-bound algorithm that solves the Three-dimensional Knapsack problem and is developed by Martello et al. [31]. This algorithm keeps track of the total item volume of the best found packing. It generates a tree in which the nodes represent a subset of items $I \subseteq J$, where J is the set of all items. At each node, multiple children nodes are generated. Each child adds a new item $i \subseteq J \setminus I$ to the packing at a feasible location. The feasible locations are determined through corner points, as explained in section 2.3.1. In short, all the children of a node are all the combinations with a newly added item at all possible locations.

Once the algorithm knows that a certain combination already leads to a higher volume compared to the best found packing, it does not need to evaluate the children since adding those will only increase the volume even more. By doing this, many combinations can be excluded and thus computation time is reduced, while still guaranteeing an optimal solution.

While the ONEBIN algorithm considerably reduces the amount of combinations to be explored, Martello et al. state that this approach is still very time consuming. The authors also expanded ONEBIN to work with multiple bins through the use of main tree branching. What they propose is to use a main branching tree to assign items to bins, without actually placing them. Then, for every bin, the ONEBIN algorithm is used to place the items at specific locations. The ONEBIN returns whether the items fit into the bin at all and if so, what the exact placements are.

2.5.2. Tree Search

Fekete et al. [15] also use tree search to find the optimal solution to a Three-dimensional Bin Packing problem. Similiarly to ONEBIN their algorithm uses a branch-and-bound method to iteratively generate combinations of items for a certain amount of bins. For each node, it uses graph theory to check whether the packing can fit. To do this, the algorithm projects the coordinates of every item within a bin onto a specific axis. Then, for every axis a graph is generated in which items are connected if overlapping occurs between their projections on that specific axis. A packing is valid if the following three constraints are satisfied:

- The generated graphs are interval graphs.
- The items that are connected within these graph can be lined up without exceeding the size limit of the axis of the graph/bin.
- No edges in the generated graphs intersect.

The lower coordinate of an item is always 0 or the upper coordinate of any previously placed item. These so called placement points are used to determine the exact placement location of an item. The algorithm iterates over all available placement points and once it finds a placement point in which the newly placed item fits, it will assign the item to that location.

2.6. Heuristics

Heuristics are inexact algorithms which generate approximate solutions to a problem. While one does not know how good this approximate is, a solution is generally found much quicker compared to exact algorithms. With good heuristics a sufficient solution should be found within an acceptable time frame.

2.6.1. Beam Search

Araya and Riff [4] created a beam search heuristic, which greedily evaluates nodes in a search graph. The authors themselves draw a comparison to a branch-and-bound algorithm (such as in 2.5.1), that only evaluates the children of the most promising nodes. Because not all nodes are expanded upon, even though their used volume is still lower than the best found solution at that time, the optimal solution might not be found. However, the computation time saved by only focussing on promising nodes could be well worth the trade off for big solution spaces.

This algorithm was created for the Three-dimensional Knapsack Problem, meaning that it tries to pack one single bin as efficiently as possible. In this problem instance, the initial node is an empty cuboid, which represents the bin. The children of a node are packings in which a single item is added. A terminal node represents a loading plan in which no more items can be added.

The algorithm starts by selecting an empty maximum space, as described in Section 2.3.3, based on the Manhatten distance. For every corner of an EMS, the Manhatten distance to the corners of the bin are calculated. The EMS with the smallest distance to any corner is chosen and if there is a tie, the EMS with the biggest volume is selected. Once an EMS is selected, w children nodes are generated, in which each child contains one of the w best fitting items to be placed. These items are placed at the corner of the EMS that is closest to a corner of the bin. How good an item placement is, is calculated by:

$$f(i, r) = V(i) - V_{loss}(i, r) \tag{2.20}$$

Where i is the to be placed item, r is the EMS, $V(i)$ is the used volume of the item and $V_{loss}(i, r)$ is the estimated wasted volume if the item is placed in r.

This strategy leads to items being first placed at the corners of a container, then the edges and finally in the middle. Besides this, the algorithm favours placing big items at the beginning and small items at the end of the packing.

Out of the resulting w children, x are expanded upon based on how promising they are. Every child is evaluate through a greedy procedure, which iteratively places an item in the bin until no item can be packed any more. Then for every child, the total volume of the packed items based on this greedy procedure is calculated and the x children with the highest volumes are expanded upon. If two children get the same items placed according to the greedy procedure, the child with the current highest load is removed. This prevents children from converging to a somewhat equal solution. This whole process is repeated until no more children can be expanded upon or a time limit is reached. At all times, the best packing solution (found so far) is saved and returned at the end.

Araya and Riff did not look into how beam search could be implement for multiple bins. However, it is trivial to allow for this. One could, for example, simply open a new bin once all other bins are full and rerun the algorithm with the remaining items.

2.6.2. Column Generation

Mahvash et al. [30] propose to use Column Generation to minimize the amount of used bins during bin packing. Column Generation is used on a mathematical model, such as the one described in Section 2.2, and tries to solve a small part of it. After solving a set of variables, it analyses the solution and if it is sufficient more promising variables are added to this partial model. This is repeated until it is proven that there are no more variables left that can be used to improve the objective function. Column Generation generally knows three problems. The master problem (MP), the restricted master problem (RMP) and the sub-problem (SP). The MP is a relaxed set-partitioning formulation of the whole model. To generate a set-partitioning formulation, Dantzig-Wolfde decomposition is used [12]. The RMP is a relaxed model of only a subset of variables. The SP is a new problem which is used to identify new (promising) variables that can be used to improve the objective function of the MP. In the SP a heuristic is used to place items. The algorithm starts by generating columns (packings). These columns are then used to solve the RMP, which obtains so called 'dual values'. These dual values are then applied to the SP and feasible packings with negative reduced cost are obtained. These packings are added to the RMP again and the process is repeated until no feasible packings with negative reduced cost or no improvement in the solution of the RMP is found. Figure 2.8 provides a high level overview of this Column Generation algorithm.

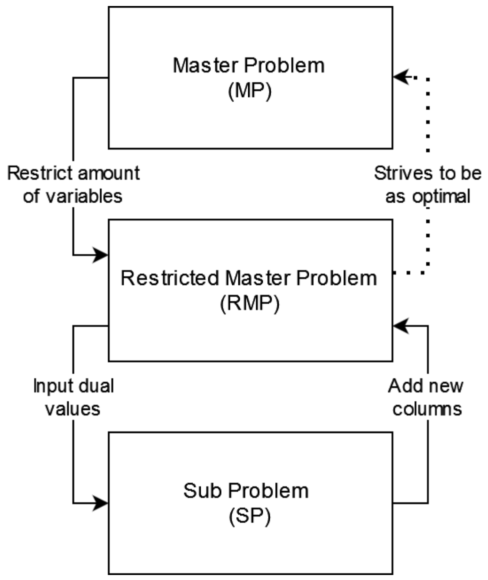


Figure 2.8: Relations between MP, RMP and SP.

The placement locations in the SP are defined by generating empty spaces at the extreme points, which are described in Section 2.3.2.

2.7. Metaheuristics

Metaheuristics are problem independent algorithms that guide the search process. As stated by Sörensen and Glover [37]: a metaheuristic generates a set of strategies to develop heuristic optimization algorithms. Metaheuristics are usually incomplete, which means that they are not guaranteed to find a feasible solution or report that no feasible solution exists within the search space. Nevertheless, the use of metaheuristics can prove to be quite useful if there is limited computation power or an immense search space.

2.7.1. Biased Random Key Genetic Algorithm

A Biased Random Key Genetic Algorithm (BRKGA) has first been introduced by Gonçalves and Almeida [18] for an assembly line balancing problem. In 2013, Gonçalves and Resende have iterated upon this algorithm to work for Three-dimensional Bin Packing [19]. This algorithm randomly packs the order and rotation of the items into a chromosome. Each chromosome contains $2n$ genes, where the first n genes are used to describe the item sequence and the $n+1$ to $2n$ genes are used for the item orientation. The algorithm starts by creating an initial population of x chromosomes, with random gene values. Then for each chromosome the following operations are performed:

1. Decode the item sequence
2. Decode the item orientations
3. Place items based on sequence and orientation
4. Evaluate packing based on fitness function

Item sequence is determined by giving every item/gene a value. Then these values are sorted in ascending order, which means that the item with the lowest value is placed first. The placement of items is based on the use of empty maximum spaces (EMS), as described in section 2.3.3, without accounting for full support. If a new EMS has a lower volume than any of the to be packed items or if its smallest dimension is smaller than any dimension of every to be packed items, it is removed.

Item placement is determined through the Distance to the Front-Top-Right Corner (DFTRC). For every feasible EMS and every feasible rotation inside that EMS, the DFTRC is calculated for the to be packed item, which will always be placed at the origin of an EMS. The origin of an EMS is always on the opposite site of Front-Top-Right Corner (thus the Back-Lower-Left corner), as can be seen in Figure 2.9. The EMS with the highest DFTRC is chosen and the item is assigned to that location. If no suitable EMS is found, a new bin is created, which is an EMS by itself.

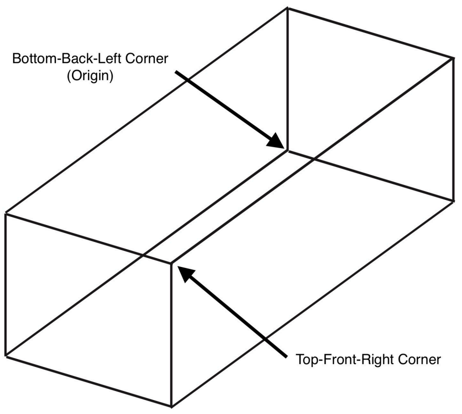


Figure 2.9: The Front-Top-Right and Back-Lower-Left corners of an EMS.

Once the placement location of an item is chosen, the orientation will be decided. Instead of choosing the rotation with the best DFTRC, Gonçalves and Resende discovered that using the following formulation lead to better results:

$$BO^* = BOs(\lceil VBO_i \times nBOs \rceil) \tag{2.21}$$

Where BO^* is the chosen orientation, BOs is the vector of possible orientations of item i in the chosen EMS and VBO_i is the value of item i in the orientation chromosome given by the BRKGA. $nBOs$ is the number of item orientations in vector BOs.

After all items are placed, the fitness of the solution/chromosome can be determined. This is done according to the following function:

$$aNB = NB + \frac{LeastLoad}{BinCap} \tag{2.22}$$

Where aNB is the adjusted numbers of bins. NB is the number of used bins, $LeastLoad$ is the load of the least loaded bin and $BinCap$ is the capacity of each bin.

What is left are x chromosomes, with each a fitness value. These chromosomes are divided into a small 'elite' group x_e (chromosomes with a relative high fitness value) and a 'non-elite' group $x - x_e$. The x_e group is copied to the new generation. A group of mutants x_m is created, which are chromosomes with completely random genes. The other x - x_e - x_m chromosomes are generated through combining genes of a chromosome of the 'elite' group with a chromosome of the 'non-elite' group. This is called crossover. The chance of obtaining a gene from the 'elite' parent (p_e) is always bigger than the chance of obtaining a gene from the 'non-elite' parent ($1 - p_e$). Figures 2.10 and 2.11 provide a visual overview of the creation of a new generation and the crossover process.

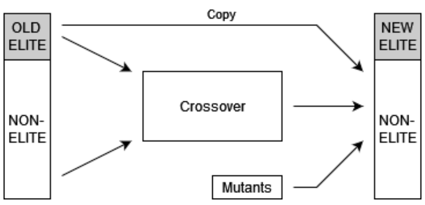


Figure 2.10: Visual depiction of new generation creation.

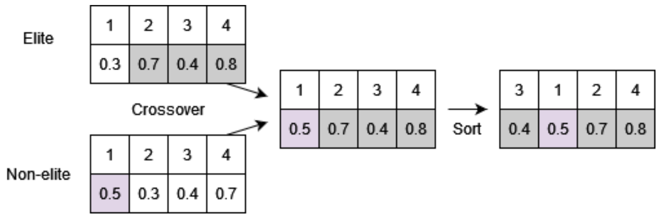


Figure 2.11: Example of crossover for the sequence of four items/genes.

2.7.2. Biased Random Key Genetic Algorithm + Variable Neighborhood Descent

A more recent paper by Zudio et al. [42] proposes adding Variable Neighborhood Descent (VND) to the BRKGA. By using their approach a better solution can be found at 100 generations compared to the solution found by the standard BRKGA algorithm at 200 generations. The authors of this paper observed that the best solutions often had the items in a packing sequence ordered by volume, width, height or length. Therefore, instead of generating random chromosomes for the initial population, they suggest to generate chromosomes in which items are sequenced in these four orders. These chromosomes are called 'universal individuals'.

The initial or mutant population starts of with one random chromosome. The best found fitness value is kept track of, which at first is the value of this random chromosome. This chromosome then does crossover with all universal individuals until either the population has reached a given size or all universal individuals have been iterated over. If during this crossover a fitness value has been found that is better than the best found fitness value, the best fitness value is replaced and the crossover iteration over the universal individuals is restarted. If no better fitness value has been found and all universal individuals have been iterated over, a new random chromosome is generated and the process is repeated until the population has reached a given size. In Figure 2.12 the total used bins per generations for both BRKGA+VND and BRKGA can be observed. As can be seen, BRKGA+VND starts off with a better solution and converges faster.

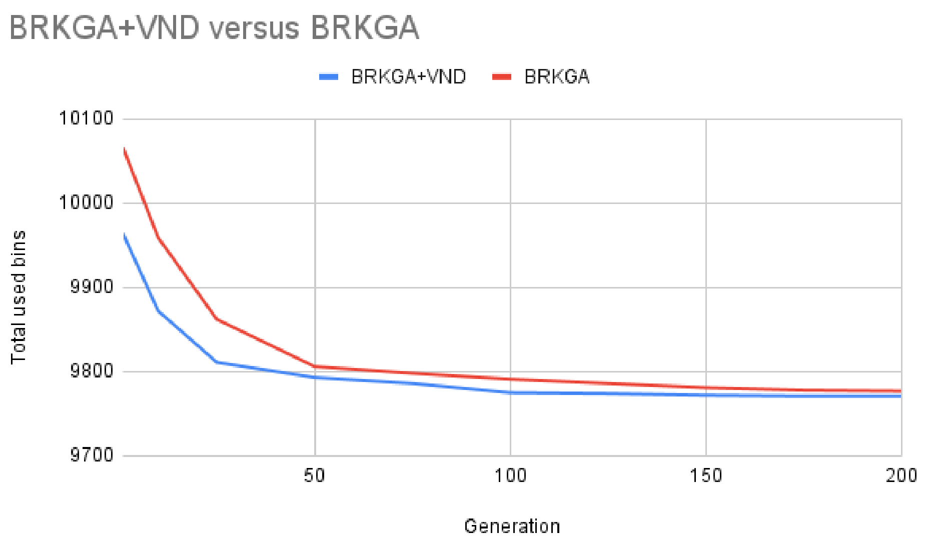


Figure 2.12: BRKGA+VND versus BRKGA over 200 generations [42].

2.7.3. Hybrid Genetic Algorithms

Feng et al. [16] propose the so called Hybrid Genetic Algorithms (HGAs) to tackle the Three-dimensional Bin Packing problem. As the name suggests, their approach contains two Hybrid Genetic Algorithms, one designed to tackle instances with a small amount of items (HGA-S) and another that is used for larger instances (HGA-L). For both algorithms, the order and rotation of the items is given by a Genetic Algorithm which uses elitism, similarly to BRKGA. One difference is that HGAs gives ‘elite’ chromosomes a higher probability to be selected for crossover, but does not guarantee that an ‘elite’ chromosome is chosen (i.e., it could be that two ‘non-elite’ or two ‘elite’ chromosomes are selected as parents). Placement locations are found through projecting points from the corners of a placed item unto other items or the bin, comparable to the extreme points discussed in Section 2.3.2. After this, the point that can fit the to be placed item and has the biggest DFTRC, as explained in Section 2.4, is chosen. HGA-S converts an entire chromosome into one packing plan, while HGA-L runs the Genetic Algorithm on each container. If HGA-L can not find a feasible packing for a container after a certain amount of generations, it will open up a new container and try again. In other words, HGA-L separates the assignment of items into bins from the placement of items in a bin, while HGA-S combines these processes in the Genetic Algorithm.

One might notice that BRKGA and HGAs seem quite similar. Both papers have been published in the same year and do not mention each other[1]. Thus, it is very likely that they have been developed in parallel without knowing of each others existence.

2.8. Comparisons

Before comparing the aforementioned algorithms, it is important to note that doing these comparisons is not as straightforward as one might think. For example, every paper has a different definition of stability, which is thus calculated in a different manner. Besides this, every algorithm has its own approach to finding and selecting placement locations, which could also influence the performance. Lastly, the data mentioned in this section is a collection of data mentioned by different papers. Because for every paper different hardware is used, direct comparisons for computation time is not possible. Nevertheless, the computation time and used hardware, as mentioned by the authors, is still discussed as to give some indication of the computational load for each algorithm. Table 2.3 lists the hardware used for each paper.

[1]While the paper of HGAs does not mention the BRKGA for Three-dimensional Bin Packing, it does mention another paper that uses BRKGA for the Knapsack Problem, which was published a year earlier [20].

Table 2.3: Hardware used during experiments of compared papers.

Paper	Hardware	
	CPU	RAM
Beam Search	2x Intel Xeon quad-processor 2.20 GHz	8 GB
BRKGA (Knapsack)	AMD Opteron 6 Core 2.2 GHz	
BRKGA (3D Bin Packing)	Intel i7-2630QM 2.0 GHZ	
BRKGA + VND	Intel i5-240CM 2.5 GHz	8 GB
Column Generation	Intel 2.20 GHz (Dell 2000)	
HGAs (HGA-S)	Pentium 4 3.4 GHz	1 GB
ONEBIN + Main tree branching	HP PA-8000 160 MHz	
Tree Search	Sun UltraSPARC 175 MHz	

2.8.1. Three-dimensional Knapsack Packing

According to Araya and Riff [4], the Beam Search heuristic leads to a slightly higher utilization rate than a Biased Random Key Genetic Algorithm variant that is adjusted to fit the Three-dimensional Knapsack Packing problem [20]. The utilization rate is the the total volume of loaded items divided by the total volume of a bin. These comparisons were done by combining the data set of Bischoff and Ratcliff [7] and Davies and Bischoff [13], which are commonly used data sets in literature.

Table 2.4: Hardware used during experiments of compared papers.

Algorithm	Performance	
	Avg. util rate	Avg. time (s)
BRKGA	92.24	232
Beam Search (150s)	92.87	150
Beam Search (30s)	92.48	30

Based on the results shown in Table 2.4, Beam Search is the better approach. However, the improvements made by Zudio et al. [42], as mentioned in 2.7.2, might close this gap and make both algorithms competitive again. Since no direct comparison has been made between BRKGA/VND and Beam Search, nothing conclusive can be said about their respective performances.

2.8.2. Three-dimensional Bin Packing

The comparisons for Three-dimension Bin Packing have been performed on a combination of instances proposed by Martello et al. [31] and Berkey and Wang [6]. This combination contains 8 classes, in which there are groupings of 50, 100, 150 and 200 items. For every grouping ten instances are generated on which the algorithm is performed. For a more detailed description of these instances, see Mahvash et al. [30]. In the tables below the average of every group of ten instances is calculated and then summed over all other groupings to get the total average amount of bins and time. An overview of the time and bins per grouping can be found in Appendix E. None of the algorithms mentioned below take stability into account.

Table 2.5: Amount of bins used by Column Generation, BRKGA and BRKGA+VND without rotation.

Algorithm	Performance	
	Total avg. amount of bins	Total avg. time (s)
LB	9487	-
Column Generation	9766	-
HGAs (HGA-S)	9769	-
BRKGA+VND	9771	526.4 (T-200)
BRKGA	9777	222.4 (aT) & 1280.8 (T-200)

Where aT is the time until the best bin packing solution was found and T-200 is the time until 200 generations (the stopping criterion) has been reached. Note that HGAs is run through HGA-S for these problem instances and that it has a time limit of 100 s per experiment. Because of the latter, total average time is not included in the table since it would give a skewed picture of the algorithm's performance. Table 2.5 shows that when rotation is not allowed, Column Generation has the best results, closely followed by HGA-S. The lower bound was obtained by Mahvash et al. [30].

Table 2.6: Amount of bins used by Column Generation, BRKGA and BRKGA+VND with rotation.

Algorithm	Performance	
	Total avg. amount of bins	Total avg. time (s)
LB (r)	8693	-
BRKGA (r)	9009	-
Column Generation (r)	9013	990.2
HGAs (HGA-S) (r)	9287	-

As can be seen in Table 2.6, when allowing for rotation, the BRKGA algorithm gets better results than Column Generation and HGA-S. No experiments have been performed with BRKGA+VND with rotation, but one could hypothesise that better results than BRKGA could be obtained. The lower bound was obtained by Boschetti [9].

The ONEBIN and Tree Search algorithms are not included in these tables because they use different instances to measure performance, namely instances with a lower item count. However, it can still be inferred that both are not competitive with the above mentioned algorithms. ONEBIN is tested by setting a time limit of 1000 s per instance and having a decision node count limit of 5000. If either of these limits is reached it returns the best found packing so far. It is observed that only 63% of the packing problems, ranging from 10 to 90 items, were solved within this time frame. Especially once more than 30 items are introduced is when the algorithm often does not find the optimal solution. Around 60 items is when the algorithm rarely finds any solution at all. When more than 70 items need be placed, ONEBIN consistently fails to find a solution. In the cases when ONEBIN did solve the problem for 50 items it took 92 s to do so, while BRKGA only needed 3.1 s. As for Tree Search, it was found that instances with more than 60 items could not be solved within the same time limit of 1000 s. Especially instances that contain a lot of small boxes would take too long to solve. These instances show, according to the authors, the limits of their implementation of the algorithm.

Of course, with more modern hardware there will be more problems that can be solved within the given time limit. However, seeing how drastically the performance drops after 30+ items for both algorithms, it seems highly unlikely that modern hardware can make them competitive given the amount of items that need to be placed in Picnic's use case. The exact amount of items is given in the confidential version of the thesis.

2.9. Conclusion

To conclude this chapter, exact algorithms have been shown to not perform adequately for instances in which the amount of items is equal to the amount of items that need to be placed in Picnic's use case. Because of this limitation on item count, a clear trend is visible in which most state-of-the-art algorithms have moved over to heuristics and metaheuristics. As of the development of this thesis, Beam Search is the best performing strategy for Three-dimensional Knapsack Packing, after which it is closely followed by BRKGA. However, Beam Search has not been proven to also work well for Three-dimensional Bin Packing. For this problem, BRKGA is still the best performing algorithm found when allowing for rotation of items. The use of BRKGA+VND while allowing for items to rotate has not been proven but might be beneficial to computation time. What also can be learned from these comparisons, as can be seen in Table 2.7, is that none of the state-of-the-art Three-dimensional Knapsack or Bin Packing algorithms incorporates all the constraints necessary for the Picnic use case. Mainly, dealing with oversized articles, fragility, rotation and bag stretching.

Table 2.7: The feature matrix for every packing algorithm mentioned in this survey.

Algorithm	Feature			
	Oversized	Fragility	Rotation	Bag stretching
BRKGA			+	
BRKGA+VND				
Beam Search			+	
Column Generation			+	
HGAs			+	
ONEBIN + Main tree branching				
Tree Search				

Because none of the state-of-the-art algorithms perfectly fit the Picnic use case and the performance of Beam Search is unclear for Three-dimensional Bin Packing, the BRKGA algorithm is chosen as a foundation for this thesis. If Beam Search performed significantly better than BRKGA, it would probably been worth the effort to convert this algorithm to handle Three-dimensional Bin Packing. However, since the differences between these algorithms are marginal, it was decided that it would be better to spent the resources needed to convert Beam Search on expanding BRKGA to fit the Picnic use case. Furthermore, the time spent on converting Beam Search could then also be used to test whether VND gives BRKGA an edge over Beam Search given Picnic's assortment and the inclusion of rotation. Because BRKGA has been proven to work well with the use of EMS to find placement locations and the generation of EMS has in practice one of the lowest time complexities (approximately the same as corner & extreme points), the decision was made to also use this strategy in combination with DFTRC. If one would want to speed up the algorithm but sacrifice accuracy, a top down matrix could be used.

Based on this literature survey, some of the research questions stated at the end of Chapter 1 can be partially answered or hypothesized about. First of all, it is now known that BRKGA can be used to pack groceries into totes and that this algorithm should be expanded upon by including oversized articles, fragility and bag stretching. How this is done exactly is explained in the next chapter. Secondly, it can be hypothesized that Three-dimension Bin Packing can lead to a fill rate higher than 85% because BRKGA reportedly has a fill rate of 92.24% for Knapsack Packing, as is shown in Table 2.4. Whether this is also true for Picnic's assortment, is discussed in Chapter 5.

Based on the literature gap found in this chapter, the scientific contributions of this thesis are:

- Extending BRKGA to pack articles that have one or more dimensions bigger than the bin it is placed within, through a single or double diagonal rotation and squeezing.
- Extending BRKGA to model bag stretching.
- Extending BRKGA to handle fragility in the form of packing zones.

- Reducing the computation time of BRKGA by using memoization, priority based multithreading and a different stopping criterion.
- A more extensive hyperparameter optimization for the hyperparameters of BRKGA[2].

A detailed description of these contributions is found in the next chapter.

[2]While the hyperparameter optimization is done for the algorithm of this thesis, its success suggests that the original BRKGA algorithm could also benefit from it.

3

3D-BAGS

This chapter serves to explain how the novel algorithm developed in this thesis operates. The proposed algorithm is a Three Dimensional Bin And Grocery Slotter (3D-BAGS). The algorithm is divided into two stages, which both are extensions of the Biased Random-Key Genetic Algorithm metaheurstic developed by Gonçalves and Resende [19]. The first stage takes all deliveries within a trip and divides them into bags. After this, the second stage divides these bags into totes. The end goal is to minimize the amount of totes used.

While the work done by Gonçalves and Resende has been used as a foundation, there are multiple modifications made to fit the Picnic use case. Furthermore, some improvements have been made to lower computation time. It is also important to note that the algorithm was originally written in C++, while Picnic requires a solution that is in Java. As of the writing of this thesis, no Java variant has been found of the full algorithm of Gonçalves and Resende. Thus, this thesis provides the first complete Java adaptation (and extension) of the prior state-of-the-art Three-Dimensional Bin Packing algorithm. A visual overview of 3D-BAGS is given in Figure 3.1. This figure denotes which features are uniquely developed for this thesis and which features have been adapted from other work.

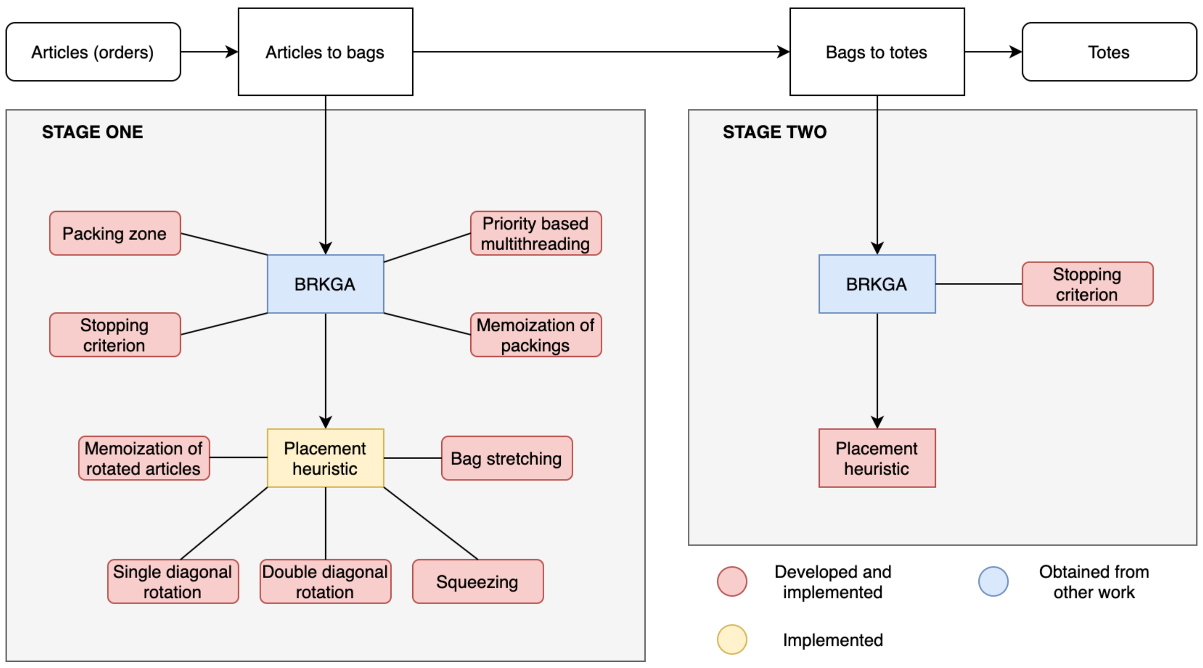


Figure 3.1: A visual overview of 3D-BAGS where features in red are uniquely developed and implemented for this thesis, yellow features are implemented based on existing research and blue features are directly obtained from other sources.

3.1. Stage one (articles to bags)

3D-BAGS starts with placing articles into bags. Instead of splitting a tote into three bags, a bags to tote strategy was chosen because with Three-dimensional Bin Packing one needs to take the exact placement location of articles into account. One-dimensional Bin Packing does not care about the placement location of each article, thus it is quite trivial to divide a tote into three bags. However, if the placement of articles must be kept intact, then a cutting mechanism is needed to cut the tote in three parts such that no bag cuts through an article and no articles of different deliveries are placed in the same bag. While multiple algorithms exist that can do these so called Three-dimensional Stock Cutting [32, 27], there are two reasons why these sorts of algorithms do not fit the Picnic use case. First of all, they take too much time to find or converge to a solution. Secondly, one could imagine that it is impossible to get a feasible solution if articles are placed in such a way that more than 3 bags are needed to separate deliveries. An example of this is given in Figure 3.2.

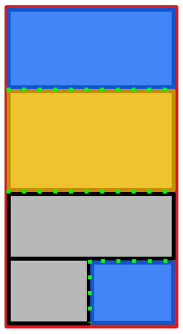


Figure 3.2: An example in which stock cutting can only be done for which three deliveries (indicated by different colours) are separated in at least four bags.

The following aspects of the algorithm are discussed in this subsection. First, the inclusion of bag stretching is discussed, after which the objective function is explained. In the third subsection, extensions to the Biased Random-Key Genetic Algorithm are discussed. The fourth and fifth subsections elaborate the mechanisms behind the placement heuristic for partially filled and empty bags respectively. The last two subsections explain how the algorithm deals with articles that do not fit into an empty bag by diagonally rotating or squeezing them.

3.1.1. Bag stretching

Currently at Picnic, the algorithm regards the volume of a bag as $\frac{1}{3}$ of a tote. However, in practice bags are stretched (or shrinked) to be more or less than this. Since a Three-dimensional Bin Packing algorithm needs to know where to exactly place an article, it should also know how much a bag is or can be stretched. Furthermore, if a bag was not allowed to stretch, a significant portion of Picnic's assortment would not be able to fit due to their long shape. It was determined that a bag can stretch up to half a tote. The bag stretch value is determined by the following formula:

$$bagStretch = ((bagLength * 1.5) - maxLength)/maxLength \tag{3.1}$$

Where $bagLength$ is the maximum end coordinate in length over all articles inside a bag. $maxLength$ is the maximum length of a bag, which is $\frac{1}{2}$ of a tote and thus dependent on the temperature zone. What can be deduced from this formula is that the value of $bagStretch$ can range from -1.0 (length of 0) to 0.5 (length of half a tote). A value of 0 means that the bag is not stretched and thus $\frac{1}{3}$ the length of a tote.

The stretch of a bag is intially set by the first article placed inside an empty bag. If this stretch is positive, it can not be changed by other subsequently placed articles. This means that all articles that are placed next, should fit in the bag given its size caused by the initial stretch. However, if the first placed article causes a negative stretch, future placed articles can reduce this stretch all the way back to 0 ($\frac{1}{3}$ of a tote). This was done because it was observed that the algorithm would create inefficient packings when a small article (e.g., a chocolate bar) was placed inside an empty bag. Such small articles would cause the bag to be so negatively stretched that no other article could fit in the bag as well, which resulted in a lot of small bags with only one article. The final stretch of a bag is determined when no more articles are packed inside it.

3.1.2. Objective function

The end goal of 3D-BAGS is to minimize the amount of totes needed to ship groceries to customers. One could say that the objective of stage one is thus to minimize the amount of bags in which the groceries are packed. While it is certainly true that the minimization of $bagCount$ is an important objective, there are a few more objectives that should be taken into account.

Firstly, Gonçalves and Resende have shown that instead of minimizing $bagCount$ one could minimize $bagCount + \frac{leastLoad}{bagCap}$ [19]. Here $leastLoad$ is the load (volume) of the least loaded bag and $bagCap$ is the maximum capacity of said bag. It was found that instead of 9806 bags, their new objective function lead to 9777 bags being needed. The idea behind this is that if two packings lead to the same amount of bags, the genetic algorithm will favour the packing in which one bag is almost empty. The hope is that the genetic algorithm will find a new packing in which the articles inside this almost empty bag are distributed over other bags and thus the bag count can be lowered.

Secondly, if all bags were $\frac{1}{3}$ of a tote, having less bags would never lead to more totes being used. However, since the proposed algorithm incorporates bag stretching, there can be cases in which more bags lead to less totes. An example is given in Figures 3.3 and 3.4 below:

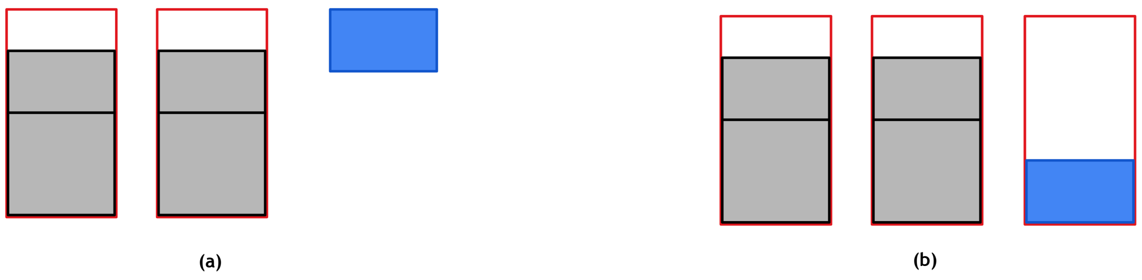


Figure 3.3: The need for three totes if there is one (normal sized) bag left.

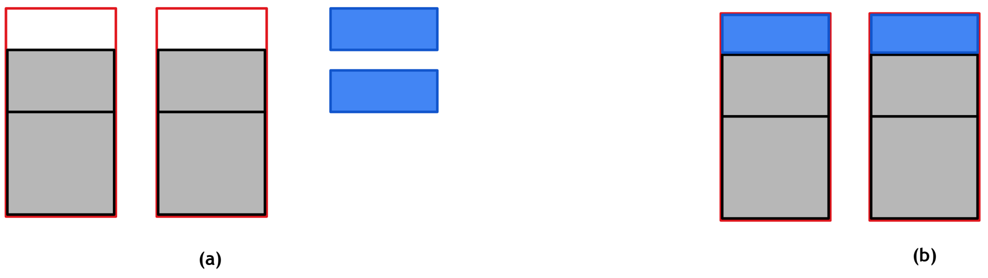


Figure 3.4: The need for two totes if there are two (small) bags left.

Because of this, the $averageStretch$ is also taken into account when calculating the objective function. The goal is to minimize this value, since if a bag is stretched to more than $\frac{1}{3}$ of a tote (positive stretch), having other bags stretched to less than $\frac{1}{3}$ of a tote (negative stretch) allows them to still fit together.

Taking everything into account, the objective function is:

$$bagCount + (w1 * \frac{leastLoad}{bagCap}) + (w2 * averageStretch) \tag{3.2}$$

Where $w1$ and $w2$ are the weights given to $\frac{leastLoad}{bagCap}$ and $averageStretch$ respectively. How the values for these weights have been determined is explained in Chapter 4.

The secondary objectives Picnic uses, which are mentioned in the confidential version of this thesis, are not implemented in 3D-BAGS. The reason is that these objectives have no additional scientific value and since they are secondary objectives they have no influence on tote count, which is the final metric for a packing.

3.1.3. Biased Random-Key Genetic Algorithm

As mentioned before, the full algorithm used by Gonçalves and Resende was written in C++. While they call this algorithm a Biased Random-Key Genetic Algorithm, it actually contains two parts: a Biased Random-Key Genetic Algorithm metaheuristic and a placement heuristic. Unfortunately, their placement heuristic knows no Java variant. But since their Biased Random-Key Genetic Algorithm metaheuristic has had many applications in different contexts, a Java variant already exists. This thesis uses the BRKGA metaheuristic developed by Mirko Alicastro, which falls under the MIT License [3]. One adjustment is made to this algorithm to fit the Picnic use case and another adjustement is discussed but ultimately not used. These adjustments are mentioned below.

Picking zone

As explained in Section 1.1.4, Picnic stores their products in picking zones, which are sorted from large, heavy and non-fragile articles to small, light and fragile articles. It is therefore important to model the chromosomes in such a way that an article is not placed before an article from an earlier picking zone. The solution to this is to look for every delivery how many picking zones are present and then order them on a scale of 0 to the amount of picking zones. For example, if there are three picking zones, the first picking zone (A) gets assigned a 0, the second (B) a 1 and the third (C) a 2. Then for every article in that delivery/chromosome, the value of its corresponding picking zone is added to its gene value. Since the gene values are always between 0 and 1, the summation with the picking zone will guarantee that the picking order adheres to the order of picking zones when sorting the chromosome, as can be seen in Figure 3.5.

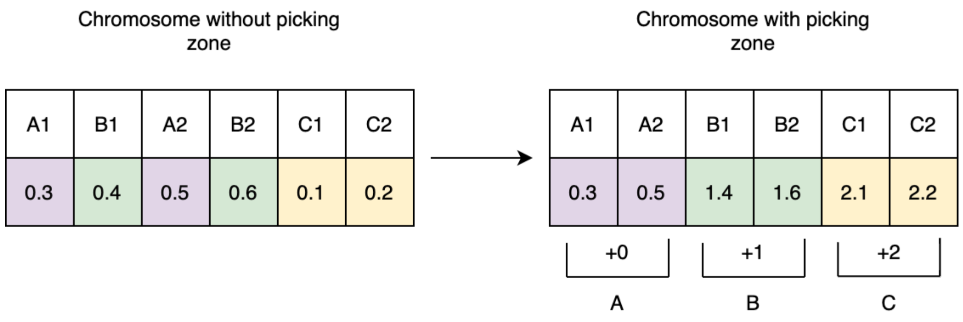


Figure 3.5: A chromosome that takes picking zone into account.

Variable Neighbourhood Descent

As mentioned in Section 2.7.2, the inclusion of Variable Neighbourhood Descent could allow 50% quicker convergence to the optimal solution [42]. While this is true for the setup used by the authors of the original paper, it was found that the inclusion of VND did not have any positive effect for the Picnic use case. The most logical explanation for this is that VND creates its universal chromosomes based on the dimensions and volume of the articles. However, since Picnic has picking zones which are already sorted by the volume of articles, the universal chromosomes do not have as big of an effect as found by Zudio et al. [42].

Also, one big downside of VND which is not mentioned by the authors, is that it does not allow for multithreading. While normally all chromosomes can be generated and then evaluated in parallel, VND needs to sequentially generate chromosomes and evaluate each one of them immediately after generation. If the evaluation indicates that the chromosome has a high fitness, it will generate more chromosomes that are a variant of this high fitness chromosome. The hope was that the reduction of necessary generations caused by VND would out weight the loss of multithreading for the initial population. However, since VND had no positive effect at all, its inclusion is not worth it.

3.1.4. Placement heuristic

To place articles in a bag, empty maximum spaces (EMS) are used to represent possible placement locations. The Distance to the Front-Top-Right Corner (DFTRC) is used to find the best suited EMS in which the article can be placed [19].

Empty maximum space generation

The generation of EMS is done through the difference process, which is proposed by Lai and Chan [27]. Every time an article is placed inside an EMS, the EMS is removed from the list of EMS in a bag and new EMS surrounding this article are generated. This generation is called the difference process because it converts the differences in occupied space between the EMS and the article into six new EMS.

Take an article with Back-Lower-Left coordinate (AX1, AY1, AZ1) and Front-Top-Right coordinate (AX2, AY2, AZ2), as depicted in Figure 3.6. The EMS in which the article is placed has a Back-Lower-Left coordinate (EX1, EY1, EZ1) and a Front-Top-Right coordinate (EX2, EY2, EZ2).

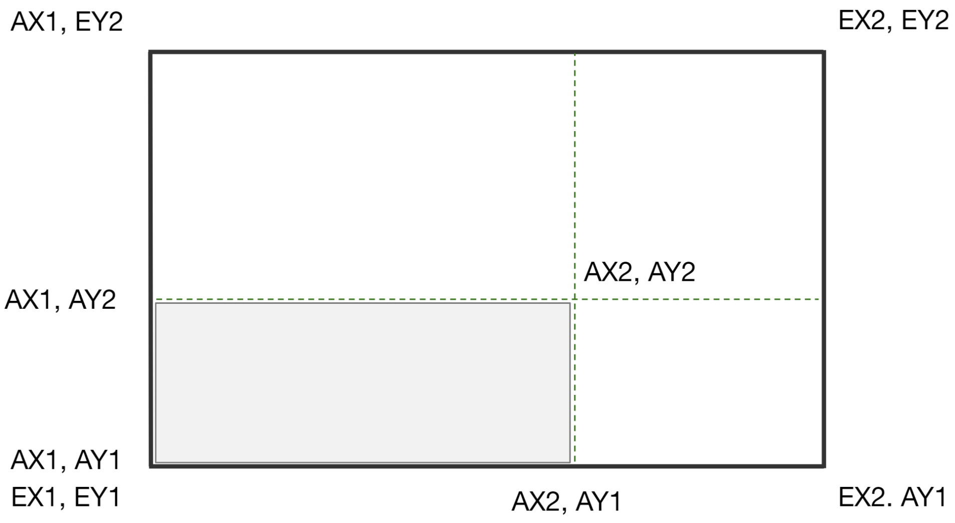


Figure 3.6: A 2d example of the difference process for an empty bag. After the placement of the article, two EMS are generated: [(AX1, AY2], (EX2, EY2)] and [(AX2, AY1), (EX2, EY2)]

The difference process generates the following six EMS in 3d:

1. [(EX1, EY1, EZ1), (AX1, EY2, EZ2)]
2. [(AX2, EY1, EZ1), (EX2, EY2, EZ2)]
3. [(EX1, EY1, EZ1), (EX2, AY1, EZ2)]
4. [(EX1, AY2, EZ1), (EX2, EY2, EZ2)]
5. [(EX1, EY1, EZ1), (EX2, EY2, AZ1)]
6. [(EX1, EY1, AZ2), (EX2, EY2, EZ2)]

After these EMS are generated, the redundant EMS are filtered out. One reason why an EMS might be redundant is if it is fully inscribed by another existing EMS. Also, if the smallest dimension or volume of an EMS is smaller than the smallest dimension or volume over all articles that are yet to be placed, it is already known that said articles can not fit into the EMS and thus keeping track of it is a waste of resources. After this filtering, the remaining new EMS are added to the list of EMS of the bag. According to Gonçalves and Resende, this will lead to an improvement in computation time of 60% [19].

Generating EMS for an empty bag is relatively simple. However, once multiple articles are put into a bag, it becomes more complex. An example of placing a third article inside a bag is given in Figure 3.7 below.

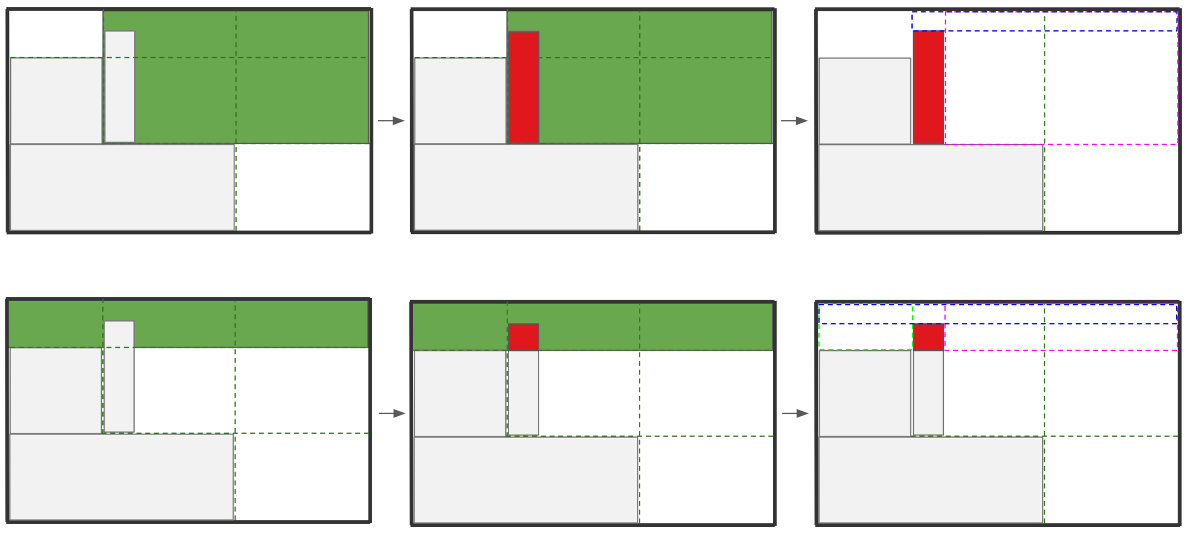


Figure 3.7: A 2d example of the difference process for a partially filled bag (containing 3 EMS).

As can be seen, even though the article is placed inside one EMS, it can intersect with multiple EMS, for which the difference process has to be performed as well. To do this, the algorithm checks for each existing EMS whether the placed article intersects it and, if so, will project the article's dimensions such that it fits into the EMS (as indicicated by the red boxes in the figure). A keen observer might also recognize that the blue EMS generated in the top right image is fully inscribed by the blue EMS in the bottom right and should thus be filtered out. Likewise, the pink EMS at the bottom right is fully inscribed by the pink EMS in the top right and should also be discarded.

Algorithm 1 Pseudocode for EMS generation when an article is placed in a selected bag

```
updatedEMSList = $\emptyset$
for each $EMS \in selectedBag$ do
    if !articleIntersectsEMS() then
        updatedEMSLIST = updatedEMSLIST $\cup$ EMS
    else
        projectedArticle = projectArticle(article, EMS)
        newEMSList = differenceProcess(projectedArticle, EMS)

        for each $newEMS \in newEMSList$ do
            validEMS = true

            if smallestEMSDimension < smallestArticleDimension then
                validEMS = false
            end if

            if EMSVolume < smallestArticleVolume then
                validEMS = false
            end if

            if validEMS then
                updatedEMSLIST = updatedEMSLIST $\cup$ newEMS
            end if
        end for
    end if
end for
selectedBag.setEmptyMaximumSpaces(updatedEMSList)
```

As stated in Chapter 2, in the worst case six new EMS are generated when an article is placed inside an EMS. Thus, the generation of EMS has a time complexity of $O(6^n)$, where n is the amount of articles to be placed. It was found that the complexity is in practice approximately $O(n^{1.55})$. This complexity was calculated by placing 200 articles in a (practically) infinitely large bag. However, for Picnic's use case, a bag is not infinitely large. Because of this, articles are spread out over multiple bags which means that the amount of intersected EMS and thus newly generated EMS is even lower for a large value of n than the approximation given in Chapter 2.

Distance to the Front-Top-Right Corner

An article is placed in an EMS that maximizes the Distance to the Front-Top-Right Corner (DFTRC).

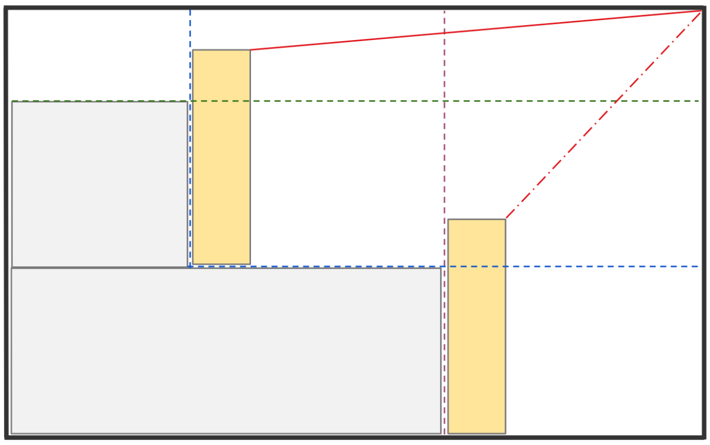


Figure 3.8: A 2d example of the Distance to the Front-Top-Right Corner calculation.

In Figure 3.8 there are three EMS, indicated by the green, blue and purple lines respectively. The yellow article has a verticle rotation, as given by the genetic algorithm. Based on this, it can be placed in either the blue EMS or the purple EMS. An article is always placed at the Back-Lower-Left Corner of an EMS. For all these possible placements the distance from the article's Front-Top-Right corner to the bag's Front-Top-Right corner is calculated, which is represented by the red lines. The red line without dashes is the longest and thus maximizes the DFTRC. Because placement at the blue EMS leads to a maximization of the DFTRC, it is chosen as the placement location for the yellow article.

To speed up the algorithm, it does not calculate the EMS that maximizes DFTRC over all bags. Instead, it iterates over all bags and selects the first bag in which it finds a suitable placement. After a fitting bag is found, the iteration is stopped and the EMS that maximizes DFTRC in that bag is chosen. The pseudocode can be found below.

Algorithm 2 Pseudocode for placing an article

```
1: selectedBag = null
2: for each $bag \in bags$ do
3:     selectedEmptyMaximumSpace = calculateDFTRC2_3D(articleDimensions, bag)
4:
5:     if selectedEmptyMaximumSpace != null then
6:         selectedBag = bag
7:         break
8:     end if
9: end for
```

If $selectedBag$ is still null after this code, it means no suitable EMS was found in any bag. After some analysis, it was found that in some cases (especially for frozen articles) the article did not fit into an EMS according to its dimensions, but could be squeezed in such a way that it would fit. A good example is bags of frites. Picnic sells a lot of different frites which are all sold in plastic bags. As one might know, these bags are very squeezable and can almost behave fluid like. So while in the Picnic database these articles all have high width and length but relative small height, they can be squeezed to be more square-like or to be other shapes.

Unfortunately, Picnic does not keep track whether an article is squeezable. It is out of scope for this thesis to go through all articles that are sold by Picnic and determine whether they are squeezable. However, determining this for a small subset of commonly bought articles can still have a significant impact. For this thesis, squeezability has been manually determined for the top 100 most sold articles per temperature zone. This determination was done quite conservatively. For example, while a bag of crisps can be squeezed, it does not behave fluid like and as such is not marked as squeezable. This conservative stance was taken because it is of upmost importance to keep the articles intact after squeezing. So anything that can be squeezed but does not behave fluid like to some degree (e.g., crisps or toilet paper) is not regarded as squeezable. A list of squeezable articles can be found in Appendix D. Unfortunately, due to confidentiality this list is only published in the confidential version of the thesis.

The algorithm will first try to find a suitable EMS without squeezing, as specified in the pseudocode above. However, if no EMS is found and the article is known to be squeezable, the algorithm will retry the DFTRC calculation but this time with the article's dimensions squeezed such that it fits inside an EMS. This squeezing is done as follows:

$$articleWidthSqueezed = min(articleWidth, EMSWidth) \tag{3.3}$$

$$articleLengthSqueezed = min(articleLength, EMSLength) \tag{3.4}$$

$$articleHeightSqueezed = articleVolume/(articleWidthSqueezed * articleLengthSqueezed) \tag{3.5}$$

Algorithm 3 Pseudocode for placing an article if it is squeezable and does not fit into any EMS without squeezing

```
if selectedBag == null & articleIsSqueezable then
    for each $bag \in bags$ do
        squeezedDimensions, selectedEmptyMaximumSpace =
         calculateDFTRC2_1D(articleDimensions, articleVolume, bag)

        if selectedEmptyMaximumSpace != null then
            placedDimensions = squeezedDimensions
            selectedBag = bag
            break
        end if
    end for
end if
```

If this code also does not find a suitable EMS, it means a new bag has to be opened in which the article can be placed. Since each EMS in all bags is iterated over and the calculation of the distance is done in constant time, the time complexity of finding the EMS that maximizes DFTRC is equal to the amount of EMS present during placement. As was shown in Chapter 2, this time complexity is in theory for the worst case scenario $O(6^n)$ but in practice approximately $O(n^{1.55})$, where n is the amount of articles that need to be placed.

3.1.5. Placement heuristic for newly opened bags

Newly opened bags can be seen as big empty maximum spaces. Thus when an article is placed inside an empty bag, the Distance to the Front-Top-Right Corner metric always puts it at the Back-Lower-Left corner of the bag. If a new bag is opened, an article is placed according to the following four steps.

1. ***Fit it***, the article is placed in the newly opened bag, given the rotation assigned by the genetic algorithm.
2. ***Force it***, the article is placed in the newly opened bag, with its dimensions forced to be parallel with the axes of the bag such that their magnitude aligns with magnitude of the width, height and length of the bag.
3. ***Rotate it***, the article is rotated diagonally as to fit into a newly opened (fully stretched) bag. If a single rotation is not sufficient a double rotation is performed. The bag is fully stretched to increase the chance of fitting after rotation.
4. ***Squeeze it***, the article is squeezed such that the oversized dimension is reduced, while other dimensions grow in order to keep volume the same. Similarly to rotation, a squeezed article is placed into a fully stretched bag, as to minimize the amount of squeezing necessary to fit. Squeezing should be done as a last resort because not all articles are squeezable and thus some squeezing can lead to damaged groceries.

With certain genetic algorithms, there is chance that a chromosome is generated which leads to infeasible solutions. For this, repair mechanisms can be used to change these chromosomes into a solution that is feasible. In the case of this thesis, the only time a chromosome leads to an infeasible solution is when an article is assigned a rotation that would not fit into a fully stretched empty bag. Step 2, 3 and 4 are used to repair such chromosomes so that the articles can fit again. In order to maximize the amount of articles that can fit in an empty bag, a bag is maximally stretched for step 3 and 4. The pseudocode is given below.

Algorithm 4 Pseudocode for placing an article into an empty bag

```
if selectedBag == null then
    selectedBag = new Bag(articleId, article.getTemperatureZone, articleLength)
end if

if !$articleFitsInEmptyBag()$ then
    placedDimensions = $calculateForcedDimensions()$

    if !$forcedArticleFitsInEmptyBag()$ then
        if $rotatedArticleInMemory(articleID)$ then
            placedDimension = $getRotation(articleID, stickOutDimensions)$
            newEmptyMaximumSpace = $getEmptySpaces(articleID)$
        else
            rotatedDimension = $calculateRotation(stickOutDimensions)$
            newEmptyMaximumSpace = $calculateEmptySpaces()$
        end if

        if !$rotatedArticleFitsInEmptyBag()$ then
            placedDimension = $calculateSqueeze(stickOutDimensions)$
            newEmptyMaximumSpace = $calculateEmptySpaces()$
        end if
    end if
end if
```

This code is run for every article that can not fit into any existing bags. In the worse case, every article placed needs a newly opened bag. Thus its time complexity is $O(n)$, where n is the amount of articles to be placed.

3.1.6. Diagonal rotation

The existing literature surveyed in Chapter 2 assumes that all items are smaller than a bag. If this was also the case at Picnic, performing only step 2 would have been sufficient to repair a chromosome. Unfortunately, Picnic also sells articles that have one or more dimensions that are bigger than half a tote (i.e., a fully stretched bag). Most of these articles should be diagonally rotated in order to fit. Two approaches have been developed to do so. First, a simple diagonal rotation is tried. If this rotation does not allow an article to fit, a more complex double diagonal rotation is performed. Both are discussed below.

Diagonal rotation

If an article is oversized, it could be that a simple rotation on the oversized axis allows it to fit, as is shown in Figures 3.9 and 3.10.

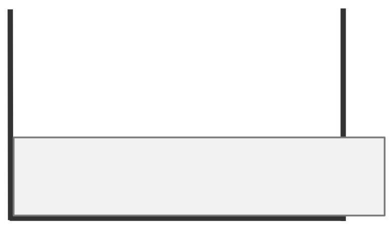


Figure 3.9: An oversized article.

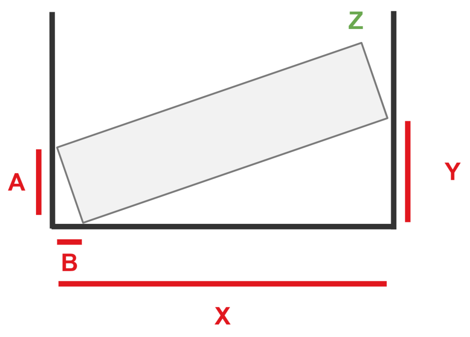


Figure 3.10: An oversized article rotated.

In order to find out what the new coordinates are of the rotated article, Pythagoras' theorem can be used. As can be seen in the figures above, in order to find the height of the rotated article, Z should be found. To do this, one needs to find Y and add the slope between A and B to it. This can be done in the following way:

$$A^2 + B^2 = H^2 \tag{3.6}$$

where H is the height of the article

$$(X - B)^2 + Y^2 = W^2 \tag{3.7}$$

where W is the width of the article

Besides this, a constraint that ensure the corners are 90 degrees must be in place such that the model adheres to the shape of a bag.

$$W^2 + H^2 = \sqrt{(A - Y)^2 + X^2} \tag{3.8}$$

What is meant with this constraint is that the diagonal between the article width and height should be the same as the diagonal between $Y - A$ and X. If this is enforced, the bag's corners are 90 degrees.

Since H (article height), W (article width), and X (bag length) are known, a Mixed Integer Quadratic Program (MIQP) can be used to find Y. For this, an Open-Source java library for constraint programming called Choco is used [36]. The following bounds are given to the solver.

Parameters	Bounds	
	Min	Max
Y	0	Article width
A	0	Article height
B	0	Article height

Table 3.1: The bounds for the MIQP solver.

Once Z has been found, new empty maximum spaces should be generated. One EMS is generated on the upper slope of the article. One below the lower slope. One above the article and one on the side (which is not visible in the 2d example below).

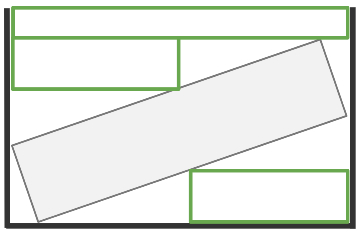


Figure 3.11: New empty maximum spaces for a rotated article in 2d.

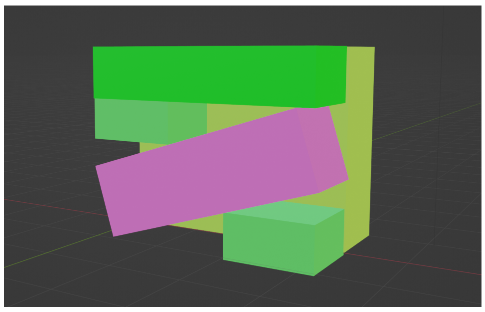


Figure 3.12: New empty maximum spaces for a rotated article in 3d.

As can be seen in Figures 3.11 and 3.12, there is some empty space that is now not occupied by an EMS. One could choose to generate more empty maximum spaces on the slope of the article, but because the algorithm does not take the effect of gravity on a slope into account, the decision was made to be a bit conservative with empty maximum spaces for rotated articles. The effect of gravity is not taken into account because it would complicate the algorithm significantly. Placing articles on a slope might make them slide down and be in different positions than they are assigned to. By being conservative with empty maximum spaces, the algorithm somewhat compensates for the slight placement differences caused by gravity.

Double diagonal rotation
In some rare instances, articles do not fit into a fully stretched bag, even after a single rotation. However, a double diagonal rotation would allow them to fit, as can be seen in Figures 3.13 and 3.14.

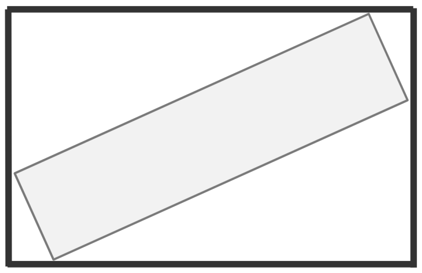


Figure 3.13: A top down view of a double diagonal rotated article.

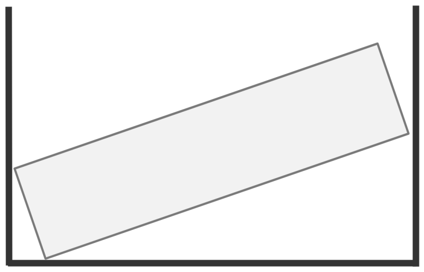


Figure 3.14: A side view of a double diagonal rotated article.

Finding the placement of a double diagonal rotation is quite more complicated than a single rotation. For this, a MIQP is used again through the Choco-solver. However this time, an article is converted into eight vertices, each representing a corner. A visual example is given in Figure 3.15.

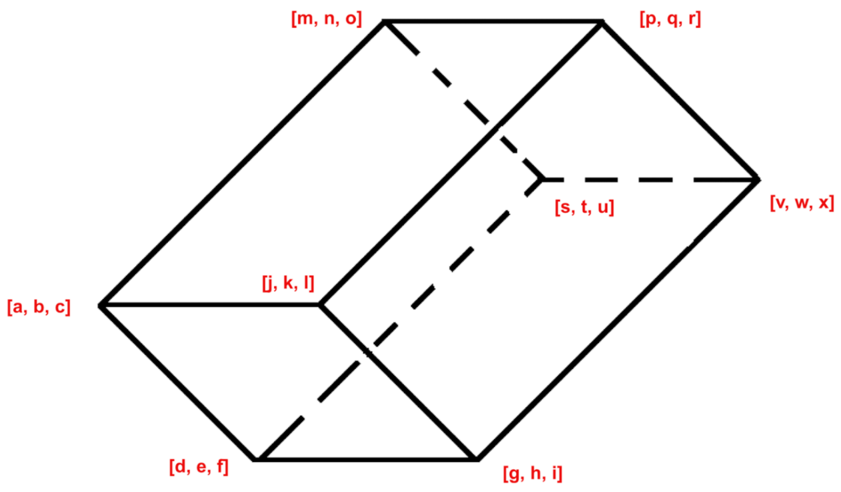


Figure 3.15: A double rotated article and its vertices.

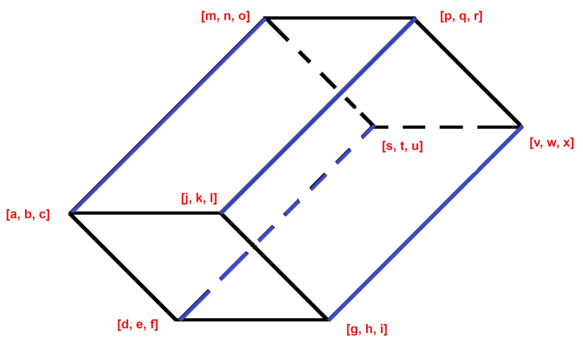
The solver is given all these vertices and the following constraints:

1. The distance between each vertex should correspond with the dimensions of the article.

2. The vertices should make a rectangular cuboid (i.e. every corner is 90 degrees).
3. Vertex [v, w, x], [s, t, u], [j, k, l] and [a, b, c] should touch the edges of the empty bag.
4. Vertex [d, e, f] and [g, h, i] should touch the bottom of the bag.
5. Every vertex must fall within the dimensions of the opened bag.

The idea behind the third and fourth constraints is that this is the most intuitive way to place a double rotated article. Of course, two corners should touch the bottom of the bag because of gravity. Then, once slightly rotated, vertices [v, w, x] and [s, t, u] will fall down as much as possible until they both hit the edges of the bag. To minimize the height of [m, n, o] and [p, q, r], [a, b, c] and [j, k, l] should touch the opposite edges of the bag. Below, a more formal description of these constraints is given.

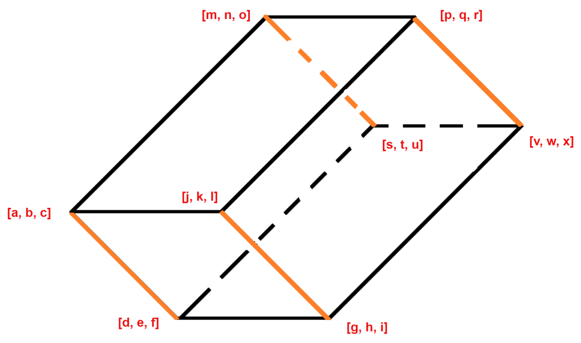
1. **The distance between each vertex should correspond with the dimensions of the article.**



articleWidth

=

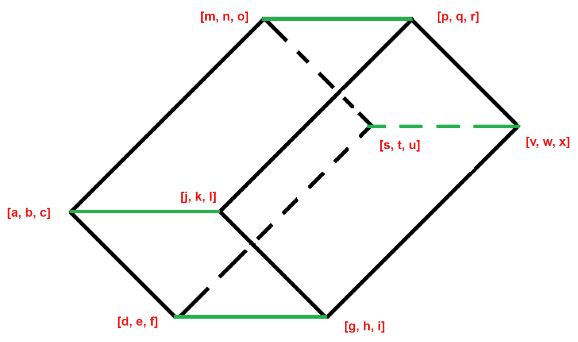
$$distance([m,n,o],[a,b,c])$$
$$distance([p,q,r],[j,k,l])$$
$$distance([s,t,u],[d,e,f])$$
$$distance([v,w,x],[g,h,i])$$



articleHeight

=

$$distance([a,b,c],[d,e,f])$$
$$distance([j,k,l],[g,h,i])$$
$$distance([p,q,r],[v,w,x])$$
$$distance([m,n,o],[s,t,u])$$



articleLength

=

$$distance([a,b,c],[j,k,l])$$
$$distance([d,e,f],[g,h,i])$$
$$distance([m,n,o],[p,q,r])$$
$$distance([s,t,u],[v,w,x])$$

Figure 3.16: The article dimension constraints for width, height and length.

2. **The vertices should make a rectangular cuboid (i.e. every corner is 90 degrees).**

   This constraint is formed by the following formula:

$$diagonalOfArticle = distanceBetweenVertices \tag{3.9}$$

   A diagonal is calculated through Pythagoras theorem:

$$diagonalOfArticle = \sqrt{articleHorizontal^2 + articleVertical^2} \tag{3.10}$$

   Where horizontal and vertical depends on the perspective (e.g., if you look at the article from the length axis, the vertical would be the article height and the horizontal would be the article width).

   The distance in the model is calculated with:

$$distanceBetweenVertices = \sqrt{(x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2} \tag{3.11}$$

   Where x, y and z are width, height and length respectively and the 1 represents the first vertex while 2 represents the second vertex.

   These constraints for the diagonals are put at two vertices, namely $[d, e, f]$ and $[p, q, r]$. If all corners at these vertices are 90 degrees, all other corners should be 90 degrees too.

$$\sqrt{articleLength^2 + articleHeight^2} = \sqrt{(a - g)^2 + (b - h)^2 + (c - i)^2} \tag{3.12}$$

$$\sqrt{articleWidth^2 + articleHeight^2} = \sqrt{(a - s)^2 + (b - t)^2 + (c - u)^2} \tag{3.13}$$

$$\sqrt{articleWidth^2 + articleLength^2} = \sqrt{(g - s)^2 + (h - t)^2 + (i - u)^2} \tag{3.14}$$

$$\sqrt{articleLength^2 + articleHeight^2} = \sqrt{(m - v)^2 + (n - w)^2 + (o - x)^2} \tag{3.15}$$

$$\sqrt{articleWidth^2 + articleHeight^2} = \sqrt{(j - v)^2 + (k - w)^2 + (l - x)^2} \tag{3.16}$$

$$\sqrt{articleWidth^2 + articleLength^2} = \sqrt{(m - j)^2 + (n - k)^2 + (o - l)^2} \tag{3.17}$$

   Note that these constraints can be simplified by removing the square roots on both sides of the equation.

3. **Vertex [v, w, x], [s, t, u], [j, k, l] and [a, b, c] should touch the edges of the empty bag.**

$$x = bagLength \tag{3.18}$$

$$s = bagWidth \tag{3.19}$$

$$j = 0 \tag{3.20}$$

$$c = 0 \tag{3.21}$$

4. **Vertex [d, e, f] and [g, h, i] should touch the bottom of the bag.**

$$e = 0 \tag{3.22}$$

$$h = 0 \tag{3.23}$$

5. **Every vertex must fall within the dimensions of the opened bag.**

$$0 \leq a, d, g, m, p, v \leq bagWidth \tag{3.24}$$

$$0 \leq f, l, i, u, r, o \leq bagLength \tag{3.25}$$

$$0 \leq b, k, t, w, q, n \leq bagHeight + stickOut \tag{3.26}$$

Where $stickOut$ is 50 for ambient, but 0 for frozen and chilled. This will be explained in the next paragraph.

A final constraint that is used is that the vertices should be placed in the same relative place as the example in Figure 3.15 (e.g., $n \geq t$ or $r \geq o$). This is not a must to find a solution, but it prevents the model from switching vertex names around, which can cause the algorithm to take the wrong vertex when calculating EMS.

Sticking out
After some talks with fulfilment centre leads, it was decided to also allow for rotated articles to stick 50 mm out of a bag for ambient totes. Frozen and chilled articles are not allowed to stick out since a lid has to be placed on top of their totes to help keeping it cold. While Picnic's algorithm does not account for ambient articles sticking out of a tote, it can occasionally happen in practice. By allowing articles to stick out 50 mm, more types of articles can fit into an empty bag without being squeezed. This makes the model more accurate because otherwise non-squeezable articles, such as long pieces of bread, would be squeezed.

Normally, the empty maximum spaces would end at the highest point of the rotated article. However, it is undesirable for empty maximum spaces to stick out of a tote as well. This is because if a small article is put in an EMS that is sticking out, there is a chance it will fall out of the bag. Because of this, the height of an EMS is limited by the height of the bag, as can be seen in Figure 3.17.

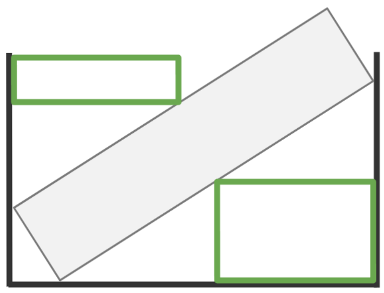


Figure 3.17: An example of empty maximum spaces for an article that sticks out of a bag.

3.1.7. Squeezing

If an article does not fit after a (double) rotation, it will be squeezed in order to fit in a fully stretched bag. This is done by reducing the oversized dimension, while simultaneously increasing the height such that the volume remains the same. It was found that no article would have an increase in height after squeezing that caused it to stick out.

It is important to keep in mind that the squeezing of an article inside an empty bag is done as a last resort as to be able to let an article fit. Thus, these squeezings are done on any article, regardless of its squeezability. The amount of articles that need to be squeezed to fit into an fully stretch empty bag is very small. Appendix C shows which articles are squeezed and whether they are squeezable at all.

3.2. Stage two (bags to totes)

In the second stage of the algorithm, the bags given by the first stage need to be distributed over totes. The goal is to minimize the amount of totes necessary to ship all the bags. In this section, first the inclusion of bag length for bag placement is discussed. After this, the objective function and placement heuristic are explained. Finally, it is elaborated when 3D-BAGS should generate multi or single-delivery totes.

3.2.1. Bag length

As discussed in Section 3.1.1, a negatively stretched bag can always have more articles added to reduce the stretch, as long as they don't stick out more than $\frac{1}{3}$ of a tote. This so called maximum placement length is used so that if a small article is placed inside an empty bag, other articles could still be placed inside that bag as well. However, when placing bags into totes, the algorithm should not place bags based on their maximum placement length. Instead of this maximum, the algorithm should take the actual length of each bag into account, which can be less than $\frac{1}{3}$ of a tote. To do this, the algorithm always keeps track of the bag length when placing articles. Every time an article is placed, it checks whether the placed article's end coordinate in length is bigger than the current bag length (given that it does not exceed the maximum placement length). If this is the case, then the bag length is overwritten to be the end coordinate of the newly placed article. Figure 3.18 provides a visual example of how bag length is updated. By using the bag length, instead of the maximum placement length, more tightly packed combinations can be generated[1].

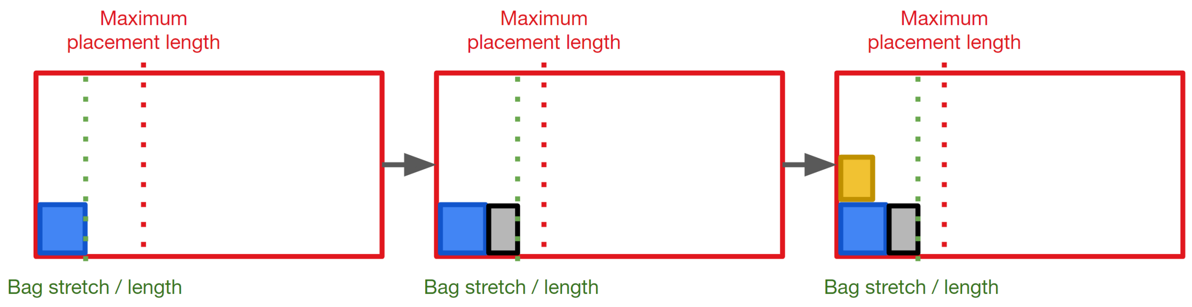


Figure 3.18: A visual example of the difference between bag stretch/length and the maximum placement length.

3.2.2. Objective function

The objective function for tote packing is inspired by the function given by Gonçalves and Resende [19]. Instead of minimizing $toteCount$, the algorithm tries to minimize $toteCount+\frac{maxRemainingLength}{toteLengthCap}$, where $maxRemainingLength$ is the biggest unused length over all totes and $toteLengthCap$ is the total length of that specific tote. By also minimizing $\frac{maxRemainingLength}{toteLengthCap}$, the algorithm favours solutions in which a tote only has a very small part of its length occupied by bags (given that tote count is the same). The hope is that these kind of totes can actually be distributed over other totes and the $toteCount$ can thus be reduced. Figures 3.19 and 3.20 provide an example in which this is the case.

[1]A keen reader might notice that bag length and bag stretch are the same. While this is somewhat true, it is important to remember that bag stretch is on a scale of -1.0 to 0.5, while bag length is simply a measurement of the length of a bag in millimetre.

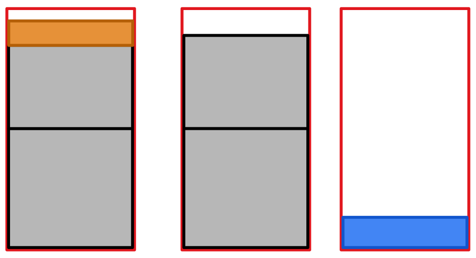


Figure 3.19: A packing in which the orange bag is packed before the blue bag. Since the blue bag does not fit in the second tote a third tote is needed.

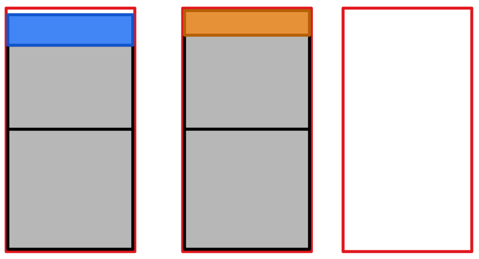


Figure 3.20: A packing in which the blue bag is packed before the orange bag. Since the orange bag (tightly) fits in the second tote no third tote is needed.

3.2.3. Placement heuristic

Firstly, a branch and bound approach was tried in combination with a first fit placement heuristic. Branch and bound is a commonly used approach to find the optimal solution in NP-hard combinatorial optimization problems and was first used by Land and Doig in 1960 [28]. Unfortunately, for this thesis it could not find the optimal solution within 10 minutes, which is far too slow for the Picnic use case. Thus, it was decided to reuse the Biased Random-Key Genetic Algorithm from the first stage, to also place bags into totes in the second stage. A few modifications had to be made in order to fit the seconds stage's needs.

Firstly, unlike articles, bags can not be rotated. So, while in the first stage a chromosome represents the placement order and rotation, in the second stage it only represents placement order. Secondly, a completely different placement heuristic is used. Since bag height and bag width is always the same in a temperature zone, a first fit heuristic is used which greedily tries to place bags into totes based on their length. A bag fits into a tote if the following constraints are satisfied:

- The bag's length is equal or less than the remaining length of the tote.
- The bag's weights is equal or less than the remaining weight of the tote.
- The tote contains less than 3 bags.
- (if the tote is single-delivery) The bags in the tote contain the same delivery id as the to be placed bag.

If no tote satisfies all these constraints for a given bag, a new tote has to be opened in which the bag is placed. When a bag is placed inside a tote, the tote is updated as follows:

$$remainingLength = remainingLength - bagLength \tag{3.27}$$

$$remainingWeight = remainingWeight - bagWeight \tag{3.28}$$

$$multiDelivery = !bagContainsOversizedArticle \,\&\, multiDelivery \tag{3.29}$$

$$toteBags = toteBags \cup bag \tag{3.30}$$

The pseudo code for the placement heuristic can be found below.

Algorithm 5 Pseudocode for placing a bag into a tote according to the first fit heurstic

```
toteFound = false

for each $tote \in totes$ do
    if tote.canAddBag(bag) then
        tote.addBag(bag)
        toteFound = true
        break
    end if
end for

if !toteFound then
    tote = new tote
    tote.addBag(bag)
    totes.add(tote)
end if
```

In the worse case, every bag has to be placed in a new tote. Before this can be done, the algorithm has to iterate over all existing totes. Thus, the time complexity of this code is $O(n)$, where n is the amount of bags that need to be placed.

3.2.4. Multi or single-delivery totes

While Picnic's algorithm generally always allows for multi-delivery totes, there are a few edge cases in which this is not the case. Picnic has manually created a list for so called 'oversized articles' which, according to the company, are articles that do not fit into a bag. If an item is oversized, Picnic puts it in single-delivery totes, since it could be placed on top of the bags of the tote. If a tote were multi-delivery, a deliverer would not know to which customer the oversized article belongs if it is placed on top of multiple bags of different customers. By forcing the tote to be single-delivery, this confusion is avoided.

In order to keep comparisons fair, the algorithm of this thesis will also force totes to be single-delivery if it contains an oversized article (as defined by Picnic). A list of oversized articles can be found in Appendix C, combined with a list of oversized articles as found and defined by this thesis. Unfortunately, due to confidentiality this list of oversized articles is only published in the confidential version of the thesis.

3.3. Performance optimizations

Using a Biased Random-Key Genetic Algorithm already leads to a solution quicker than brute force. However, there still some optimizations that have been implemented to make the algorithm faster. This section discusses five improvements made to the algorithm that lead to a decrease in computation time: multithreading, memoization of chromosomes, memoization of diagonally rotated articles, a stopping criterion and the prevention of generating EMS when the last article is placed inside a bag.

3.3.1. Multithreading

Multithreading allows for specific calculations to be done in parallel, which can speed up the computation process. Gonçalves and Resende claim that with multithreading the evaluation of chromosomes can be almost linearly sped up with the amount of cores that is given to the algorithm [19]. However, in the case of this thesis, even more parts of the algorithm can be multithreaded. On the highest level, every trip can be packed in parallel. Then for each trip, the deliveries can also be packed into bags at the same time. Within these deliveries, each temperature zone can be packed on different threads. Besides parallelizing the evaluation of chromosomes, the Biased Random-Key Genetic Algorithm can also be multithreaded during the generation of chromosomes. A visual overview of all the parts of the algorithm that can be multithreaded is given in Figure 3.21.

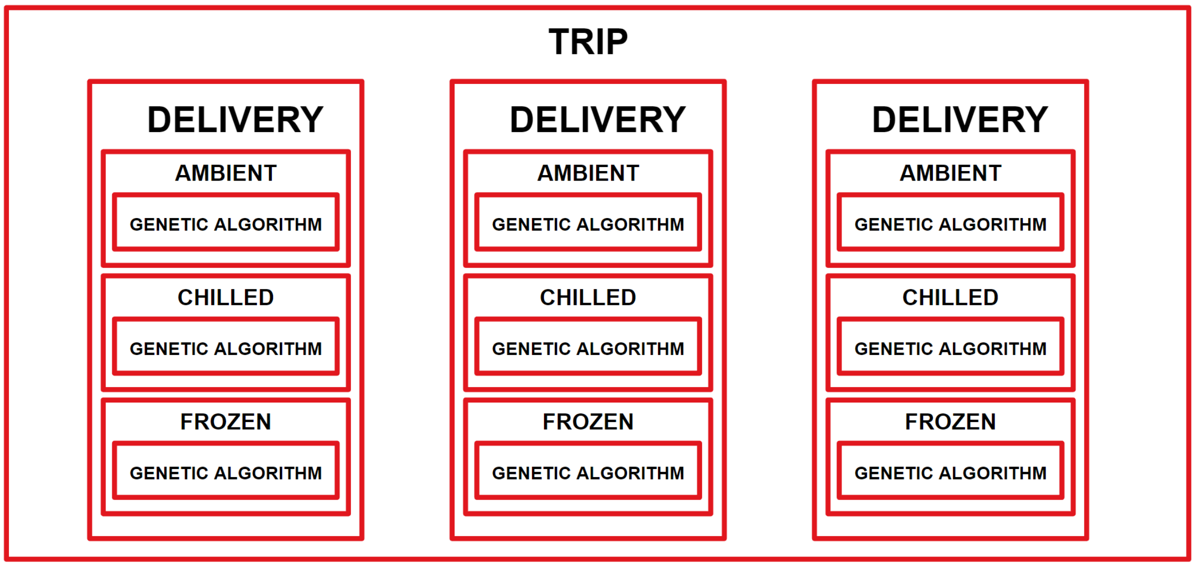


Figure 3.21: Visual overview of components in the algorithm that can be multithreaded.

Ideally, every part of the algorithm should be multithreaded. However, the effectiveness of multithreading is limited by the amount of cores available. Too much multithreading could even lead to so much overhead that there is a negative effect on performance. To prevent this, a choice has to be made on which parts to multithread and which parts to calculate in sequence. For this, a tradeoff has to be made. If multithreading is done on a high level (e.g., per trip), there is little overhead. However, imagine if one trip takes significantly longer to process than the other trips. If there is only multithreading at the highest level, a lot of cores will be unused when this final trip is still processed while all other trips are already done. Having multithreading at the lowest level will have less waiting for other processes to be finished, but there is also more overhead because more threads have to be created, deleted and kept track of.

To get the best of both worlds, the algorithm of this thesis uses prioritized multithreading. What this means is that initially there is only multithreading at a high level. But if there are threads idle, a low level process can use them. More concretely, the algorithm multithreads per delivery and if there is an idle thread, the genetic algorithm can use it to generate or evaluate chromosomes. It was chosen to use deliveries as the highest level instead of trips to keep comparisons for the results fair. All comparisons in Section 4 and 5 are done per trip. If there was multithreading per trip, then the final few remaining trips would have idle threads which could be used by the genetic algorithm. This would speed up the computation time of these remaining trips which would make the comparison of their computation time unfair compared to other trips in which no multithreading on a lower level occurs. By having multithreading per delivery, it can be made sure of that every trip has access to all available cores/threads and thus a more fair comparison can be drawn. The effect of multithreading on the performance of the algorithm can be found in Chapter 5.

3.3.2. Memoization of chromosomes

Memoization is the caching of the output of a function, given specific input. When a certain input is given, for which the output has already been computed before, the algorithm simply returns the cached result instead of having to compute the output again. An intuitive way to use memoization is to look whether a specific chromosome has already been packed before. If the chromosome is found in the cache, the stored result can be returned. However, given that in this algorithm chromosomes are made out of non-integer values, it is rare to get chromosomes that are exactly the same. Also, it could be that chromosomes with vastly different values lead to an identical packing. An example of this is given in Figure 3.22. In this figure the values in white boxes are the article id's and the colored boxes are their corresponding gene values. On the left, one can see two chromosomes before their genes are sorted and on the right the placement order after gene sorting is shown

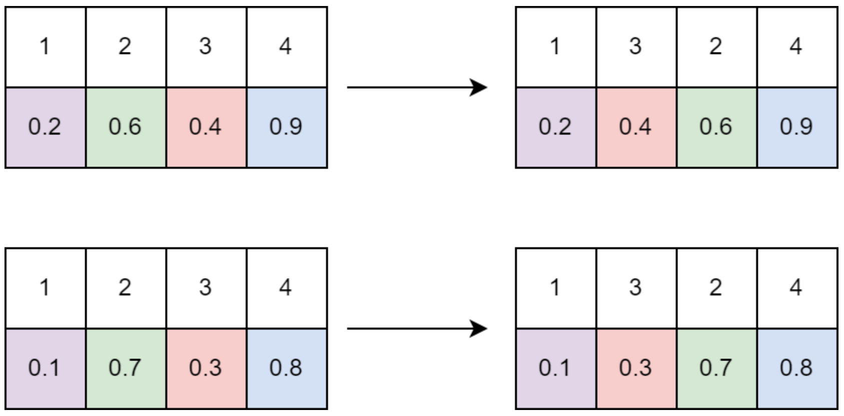


Figure 3.22: Two different chromosomes with the same packing (rotation has been excluded from the chromosome for simplicity).

The algorithm exploits the fact that different chromosomes can lead to the same packing, by first decoding chromosomes into a very simple string. This string is a concatenation of all articles in a chromosome represented by the articles' id and rotation. The order and rotation of the articles in this string correspond with the order and rotation given by the chromosome. The cache maps these strings to a value, which is the resulting fitness value of the packing. Since different chromosomes now lead to the same string if their packing is identical, they can get the results from the cache if an identical packing was already performed before. The performance impact of the memoization of chromosomes can be found in Section 5.4.4.

A risk of memoization is running out of memory for the cache. However, in practice, no delivery has been found that is so big that it leads to the algorithm running out of memory.

3.3.3. Memoization of diagonally rotated articles

While the calculation of a single rotation can be done quite fast, solving the MIQP for a double rotation inside an empty bag can take up to 83 seconds in the worst case scenario. To help overcome this long computation time, memoization is used to store the rotation and corresponding empty maximum spaces in a file that can be read and written to on different runs. What can be seen from the data is that there are not a lot of unique articles that need to be rotated to fit in an empty bag, which means that it is feasible to permanently store their rotation. Only when a new article is encountered, that never has been seen before in different runs, the program can be significantly slowed down to calculate this new rotation. Picnic could circumvent this by running the rotation algorithm separately on their whole article data set, but for the scope of this thesis it is sufficient to simply do this within the bag packing code. Using memoization leads from rotation taking a significant part of the computation time, to practically reducing its impact to 0% if it has been rotated before. The exact performance impact of the memoization of diagonally rotated articles can be seen in Section 5.4.5.

3.3.4. Stopping criterion

The Biased Random-Key Genetic Algorithm (BRKGA) uses generations to converge to a solution. In the case of this thesis, the more generations are used, the less totes are needed for a delivery (until the algorithm converges). The original BRKGA algorithm, which this thesis expands upon, found that stopping after 200 generations was the best balance between performance and computation time. While this is generally a good strategy, this thesis improves the stopping criterion by adding some more criteria.

Firstly, the amount of generations is reconfigured based on hyperparameter tuning as explained in Chapter 4. Secondly, a new criterion is added which stops the algorithm if no improvement has been found after a certain amount of generations. Since some deliveries actually converge quite quickly, it is a waste of computation time to keep iterating until the final generation. After how many generations without any improvement the algorithm should stop iterating, is also defined in Chapter 4.

Lastly, a time limit has been set that prevents the algorithm from iterating any further. This is done to deal with extreme outliers that represent immense deliveries. For example, in the data this thesis uses, a delivery contains on average $\mu = 88.83$ articles with a standard deviation of $\sigma = 42.66$.[2] However, the biggest outlier contains 1741 articles as can be seen in Figure 3.23.

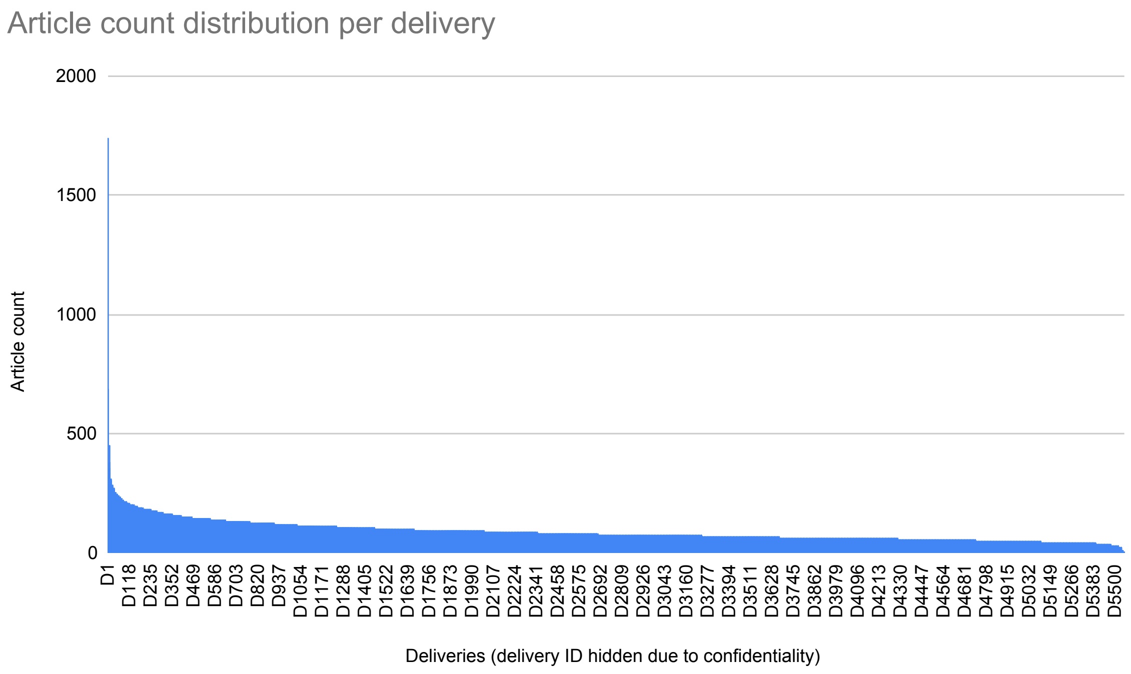


Figure 3.23: The distribution of article count over deliveries. Due confidentiality the article count is multiplied with a hidden factor and the delivery ID is hidden.

The BRKGA algorithm does not scale well to that many articles and as a result takes too much time to converge to a good solution. By limiting the algorithm by some time, we can guaranteed some solution, even if it is not the most optimal. Luckily, such big deliveries are extremely rare and thus not that big of an issue. For this thesis, the time limit is set to 50 seconds. However, Picnic might increase or decrease this value dependent on their hardware and time budget.

3.3.5. Prevention of EMS generation for the last placed article

Because of the way empty maximum spaces are generated, the algorithm becomes slower the more articles are placed. This is because after every placed article, more empty maximum spaces are generated and thus need to be checked for intersection in a later stage. When placing the final article, the algorithm does not need to update the empty maximum spaces in that bag, since afterwards no other article has to find a suitable placement. By not generating empty maximum spaces after the final placement, some computation time can be saved.

3.4. Summary of 3D-BAGS

To summarise, by applying BRKGA to a two-stage algorithm it can be used to assign articles to bags and place these bags into totes. In order to work for the Picnic use case, it has to be extended with the following features:

- Packing zones
- Single and double diagonal rotation
- Article squeezing

[2]Due to confidentiality, these numbers have been multiplied with a hidden factor.

- Bag stretching

To place articles into bags, a new approach is used which first places the articles with their provided rotation. If this fails, it will force the article to rotate such that its dimensions are aligned with the bag based on their magnitude. If the article can still not fit, it is diagonally rotated. As a last resort, the article is squeezed. In order to place the generated bags into totes a first fit approach, given the length and weight of the bags, is used.

Furthermore, by using memoization, (priority based) multithreading and a different stopping criterion, computation time can be decreased.

4 Hyperparameter optimization

Hyperparameter optimization is the process in which the set of hyperparameters is found that can lead to the best performance of a model. In the case of 3D-BAGS, performance is measured through the average tote count needed to ship groceries to customers. However, since Picnic has to fulfill many deliveries on a daily basis, the algorithm can not take too long to process a single delivery. Thus, an extra objective for the hyperparameter optimization is added which minimizes the time spent on a delivery. These two objectives conflict with each other since giving a genetic algorithm less time/iterations often leads to worse results (in this case a higher tote count). The goal is to find a set of hyperparameters which leads to the lowest average tote count, while still having an acceptable value for average time spent.

4.1. Setup

The hyperparameter optimization is done through a virtual machine provided by TU Delft. This virtual machine has an Intel Xeon Gold 6148 CPU @ 2.40GHz with 16 cores and 32 GB of memory. The hyperparameter optimization is done with the use of Optuna, which is an automatic hyperparameter optimization framework [2]. Optuna was chosen because it vastly simplifies the process of hyperparameter optimization and includes the Multi Objective Tree-structured adaptive Parzen Estimators (MOTPE), which is the best fitting state-of-the-art hyperparameter optimization technique found for this thesis, as is explained in Section 4.4.1. Unfortunately Optuna only works in Python, while 3D-BAGS is written in Java. To have Optuna work with Java a REST API is used that allows the Optuna framework to transfer a set of hyperparameters through a local connection to 3D-BAGS, which in turn sends the average tote count and average time spent back to the framework.

4.2. Hyperparameters

In order to optimize the hyperparameters, a lower and upper bound has to be set for which the optimizer can explore. These bounds define the search space in which the optimizer has to find the best performing sets of hyperparameters. A good balance should be found such that this search space is big enough to find the best performing hyperparameter sets, but small enough such that it can do this in a reasonable time. The values and reason behind the lower and upper bounds for each hyperparameter are given below.

Table 4.1: Hyperparameters for objective function weights.

Objective function	Bounds	
	Min	Max
Least load weight	0	3
Average stretch weight	0	3

Table 4.1 provides the hyperparameter bounds for the objective function weights. A bound between 0 and 3 has been chosen for both these objective function weights because of the following reasons. First of all, a lower bound of 0 makes sense as it allows the optimizer to essentially disable the weight to see if it has any effect on average tote count at all. The upper bound was chosen more arbitrarily. It was decided that it should be at least higher than 1, since the weight would then be able to become more important than $bagCount$. This means that a solution could be seen as better, even though it has more bags (but less least load or average stretch). It was deemed unrealistic that the weights could be much more important than bag count, thus a value of 3 seemed high enough to allow some increase in importance, but not too much.

Table 4.2: Hyperparameters for the Biased Random-Key Genetic Algorithm (BRKGA).

Genetic algorithm	Bounds	
	Min	Max
Population size multiplier	20	50
Elite fraction	0	0.5
Mutant fraction	0	0.5
Crossover probability	0	1

The bounds of the hyperparemeters of the genetic algorithm can be found in Table 4.2. For the population size multiplier, Gonçalves and Resende tested values in the interval of [10, 20, 30, 40] [19]. During testing of 3D-BAGS, it was found that the value of 10 often caused populations to be so small that they did not have enough chromosomes to converge to a good solution. Thus, it was decided to raise the lower and upper bound by 10 compared to the work by Gonçalves and Resende. A lower bound of 0 was not chosen because that would lead to no chromosomes being generated, which would mean that 3D-BAGS could not function at all. Elite fraction, mutant fraction and crossover probability have a lower bound of 0 in order to let the optimizer be able to disable these features completely. A upper bound of 0.5 for elite and mutant fraction was chosen since the sum of the fractions for both elite and mutant fraction could not be above 1. Crossover probability has an upper bound of 1 since the probability to inherit a gene from a specific parent should be between 0 (no probability) and 1 (guaranteed probability).

Table 4.3: Hyperparameters for the stopping criterion.

Stopping criterion	Bounds	
	Min	Max
Max iterations	150	300
Max iterations without improvement	10	50

There are two stopping criteria, of which the bounds are provided in Table 4.3. Initially, both these stopping criteria had a lower bound of 0. However, it was found that the optimizer would mostly explore values that lead to very low values for average time spent. Since there was some headroom for this objective value, it was decided to raise the lower bound as to force the optimizer to find solutions that take a bit more time (but also generated better packings). Data showed that on average, the genetic algorithm converges somewhere around 200 generations. It was thus chosen to have the lower bound for max iterations be slightly below it. A value of 300 was chosen to see whether a big trade off in computation time would be worth the slight reduction in tote count (since most deliveries would already have converged).

Max iterations without improvement has a lower bound of 10 since it is desirable to give the algorithm some room to stall while converging. A maximum of 50 was chosen since the data has shown that the algorithm rarely converges any further if it hasn't seen an improvement in 40 generations. Setting the maximum to 50 instead of 40 allows the optimizer to see whether there is indeed no gain from allowing more generations without any improvement.

4.3. Stabilization of the coefficient of variation

Each customer at Picnic has different needs and preferences. Thus, every order is quite different in its content. This means that for certain hyperparameter values, some orders can benefit while other orders might be worse off. It is very unlikely to find universal hyperparameters that are ideal for every single order. So, instead of looking at specific orders, one should find hyperparameters that on average perform best over all orders Picnic receives daily. Unfortunately, it is computationally infeasible to do hyperparameter tuning on all orders known to Picnic. Thus, a metric should be used to find a sample set that accurately represent the whole set of orders. To be able to decide whether a sample set is a good representation of all orders Picnic receives, the coefficient of variation is used. This coefficient is calculated as follows:

$$C_v = \frac{\sigma(o)}{\mu(o)} \tag{4.1}$$

Where o is the output of the algorithm, $\sigma(o)$ is the standard deviation and $\mu(o)$ is the mean.

The coefficient of variation is calculated every time a trip is processed and takes all previously processed trips into account. It is calculated for both tote count and time spent. The idea is that once this coefficient stabilizes for both these objectives, a good estimation of the algorithm's output is generated and no further trips need to be processed. Figure 4.1 shows an example of how the coefficient stabilizes over time.

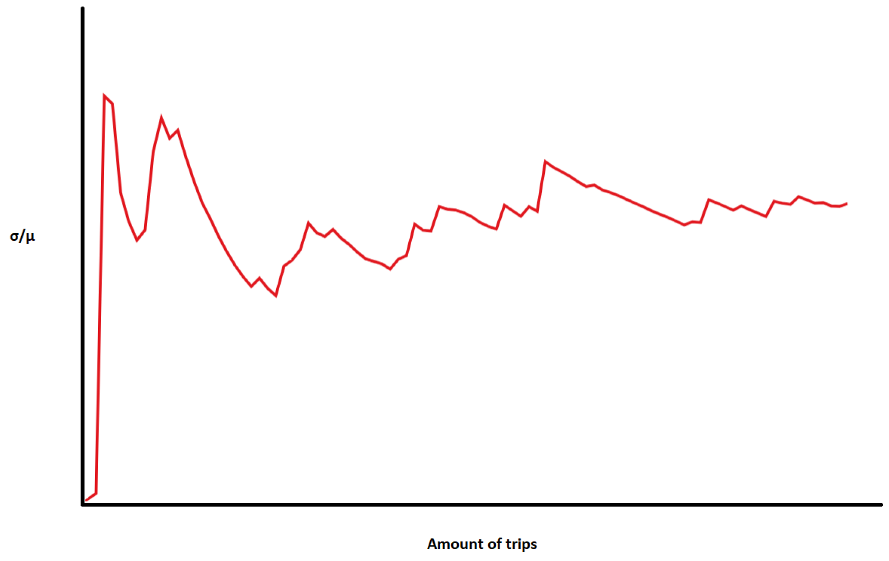


Figure 4.1: An example of the stabilization of the coefficient of variation.

One can see that there is a repeating pattern of spikes that become smaller the more trips are processed. These spikes represent trips that have a high variance. The reason these spikes get smaller over time is because $\frac{\sigma}{\mu}$ takes in all previous trips. Meaning the more trips are processed the less an effect a new trip has on the total standard deviation or mean. The coefficient of variation is not guaranteed to stabilize after the same amount of trips for different runs. Thus, tote count and time spent should be averaged by the amount of trips required for said stabilization in order to be able to draw fair comparisons.

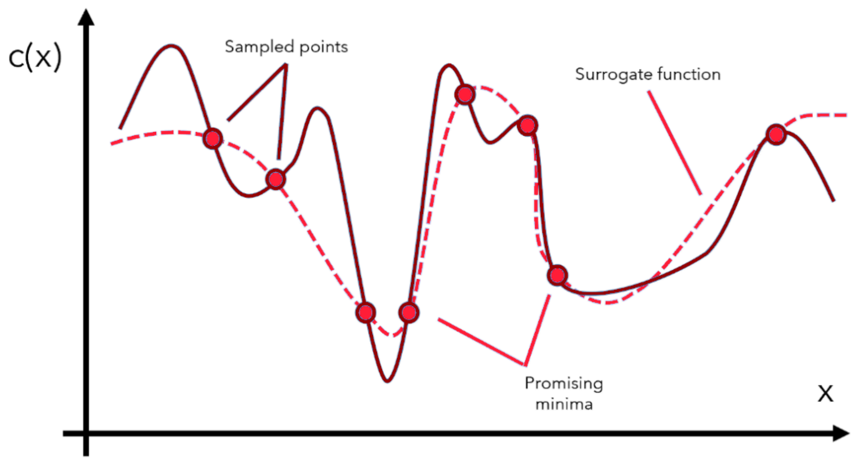
4.4. Methods

4.4.1. Multi-objective optimization

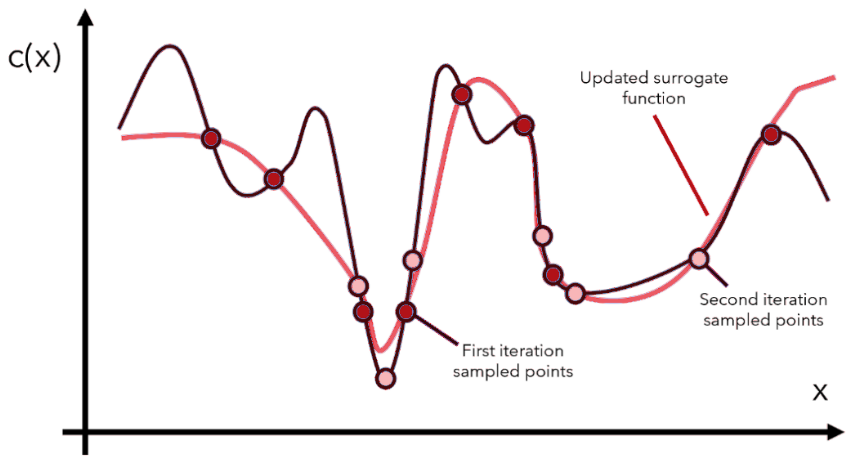
There are multiple approaches to multi-objective hyperparameter optimization. The simplest method is to try every possible combination of hyperparameters and select the best performing combinations. This is however too computationally intensive and thus often not used for optimization. One could also do a random search which randomly selects a few combinations and selects the best solutions. However, this is a quite naive approach and has a high chance of missing a near optimal set of hyperparameters. Another method is grid search, which is also done by Gonçalves and Resende [19]. Grid search means that an interval of values is decided for each hyperparameter and every combination for these values between the hyperparameters is tested. While grid search is easy to implement and can lead to good results, it has a few disadvantages. Since it uses intervals, it can miss the best values if the steps in these intervals are too big. More importantly, grid search experiences the curse of dimensionality. One can imagine that if two variables with both an interval of size 5 are tested, there are 25 combinations. However, when there are three variables, this number goes up to $5 * 5 * 5 = 125$ and $5^4 = 625$ when there are four variables. This so called combinatorial explosion makes it unfit for this thesis' use case since there are 8 variables with quite large intervals.

Another popular algorithm is NSGA-ii, which was designed by Deb et al. [14]. This algorithm is, just like the Biased Random-Key Genetic Algorithm used in this thesis, an evolutionary algorithm that uses elitism to converge to a chromosome (set of hyperparameters) with the best objective values. While NSGA-ii can lead to good set of hyperparameters, it requires a substantial amount of generations to do so. As explained before, many deliveries need to be processed during optimization in order to get a good estimation of the model's output under a set of hyperparameters (i.e. the coefficient of variation stabilizes). This means that calculating the objective value for one chromosome is computationally intensive. Since NSGA-ii requires many chromosomes per generation and many generations to converge to a good set of hyperparameters, it is not a good fit for this thesis.

Because one evaluation of a set of hyperparameters is computationally intensive, it is interesting to look at Bayesian optimization [17]. This optimization method makes use of so called surrogate functions, which are functions that are an approximation of the objective function but take significantly less time to evaluate. Bayesian optimization starts with sampling a few points from the original objective function to create a first approximation. Based on this approximation, some minima can be found. Around these minima more samples are gathered from the original function and the approximation is updated accordingly. After doing this for some time, all minima are fully explored and the global minima is likely to be found. It is worth noting that convergence to a global minima is not guaranteed. For example, if the objective function has a very erratic shape, the global minima can be missed during the sampling process. For a visual explanation of how Bayesian optimization works, see Figure 4.2.



(a) First sampling of the objective function.


c(x)
Updated surrogate function
Second iteration sampled points
First iteration sampled points
X

(b) Second sampling of the objective functino.

Figure 4.2: A visual explanation of Bayesian optimization [41]

Bergstra et al. [5] suggest the use of Tree-structured adaptive Parzen Estimators (TPE) as the surrogate instead of Gaussian process regression that is normally used. The use of TPE allows for better scaling to more variables, has a lower computational complexity and allows for more than only continuous variables (e.g. discrete and conditional). A more recently paper published by Ozaki et al. [33] expands upon the TPE and introduces the Multi Objective Tree-structured adaptive Parzen Estimators, which outperforms other state-of-the-art multi-objective optimizers in most instances with a limited time budget.

4.4.2. Multiple-criteria decision-making

As stated before, there are two objectives when doing hyperparameter tuning: average tote count and average time spent. When plotting these two objective values based on a given hyperparameters set, one notices that there is no single hyperparameter set that minimizes both objectives. An example of said plot can be found in Figure 4.3.

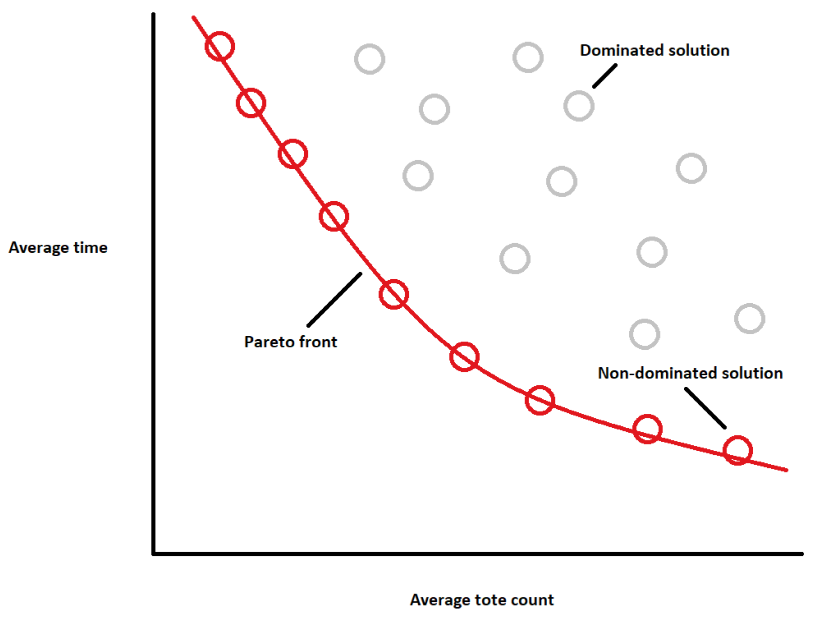


Figure 4.3: An example of a Pareto front.

This figure shows that there are some so-called non-dominated solutions. A solution is non-dominated if one objective can not be improved without degrading another objective. The set of non-dominated solutions represent the Pareto front, which are thus the best solutions with regards to either average tote count or average time spent. If one would choose the solution at the top left, it would mean that average tote count would be low but average time spent would be relatively high. In contrary, the solution at the bottom right represents a solution with a low average time spent, but a high average tote count. Now, the question is, which solution should be taken as the final set of hyperparameters on which the algorithm will run? This depends on how much weight each objective has. If, for example, one has practically unlimited time, the solution at the top left would make more sense. However, if both time and tote count weight equally, the solution in the middle of the 'knee-like' shape could be a good choice (assuming the axes are normalised).

Choosing which set of parameters to use is called multiple-criteria decision-making (MCDM) or multiple-criteria decision analysis (MCDA). It is hard to find the definitive set of hyperparameters for 3D-BAGS to operate on. One of the reasons for this is that the effect of hyperparameters is dependent on hardware. If 3D-BAGS is run on extremely fast hardware, it can probably afford to choose a set of hyperparameters that is more computationally intensive to use. A second reason is that it has been quite hard to pinpoint the time budget Picnic has for the bag and tote packing process. While their current approach runs quite fast, there is room for a slower algorithm if it means less totes are used in the end. How much headroom there is for a slower algorithm, is not precisely defined by Picnic.

Because of the above mentioned reasons, it was decided to create three configuration: quality, balanced and performance. Quality is the configuration with the lowest tote count but the most time spent. Balanced is the configuration with an average tote count and also an average amount of time spent. Performance represents the configuration with the highest tote count, but the lowest of time spent. A configuration can then be chosen by users of 3D-BAGS based on their needs.

4.5. Optimization results

For the hyperparameter tuning, it is recommended to let the Multi Objective Tree-structured adaptive Parzen Estimator run 250 trials of which the first $11 * numberOfVariables - 1$ trials contain randomly sampled hyperparameter sets [33]. Since there are 8 variables, a total of 87 hyperparameters sets are randomly chosen. After this, the MOTPE explores the minima until the 250th trial, which in this case is for the remaining $250 - (11 * numberOfVariables - 1)$ = 163 trials. Figures 4.4

and 4.5 show that the optimizer slowly converges to smaller values for both average tote count and average time spent during the exploration of the minima.

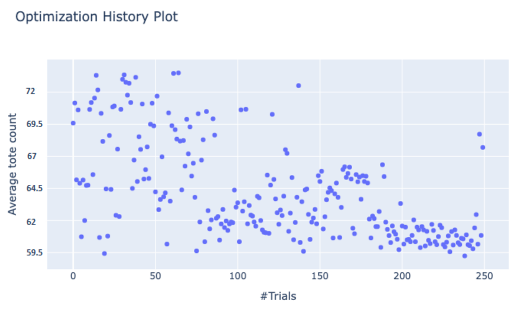


Figure 4.4: The optimization history plot for average tote count. Due to confidentiality these numbers are multiplied with a hidden factor.

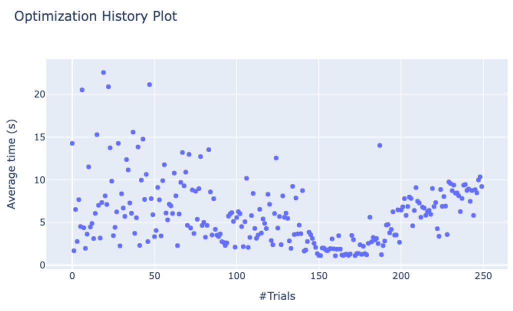


Figure 4.5: The optimization history plot for average time spent (in seconds).

As can be seen in both graphs, average tote count clearly converges to a value slightly above 59.5, while average time spent first converges to around 1 second but then increases again. What this probably indicates is that a further decrease in average tote count requires more time to be spent, which seems logical.

After running 250 trials, a Pareto front can be plotted to get some idea of how average tote count behaves in relation to average time spent. The following Pareto front has been found:

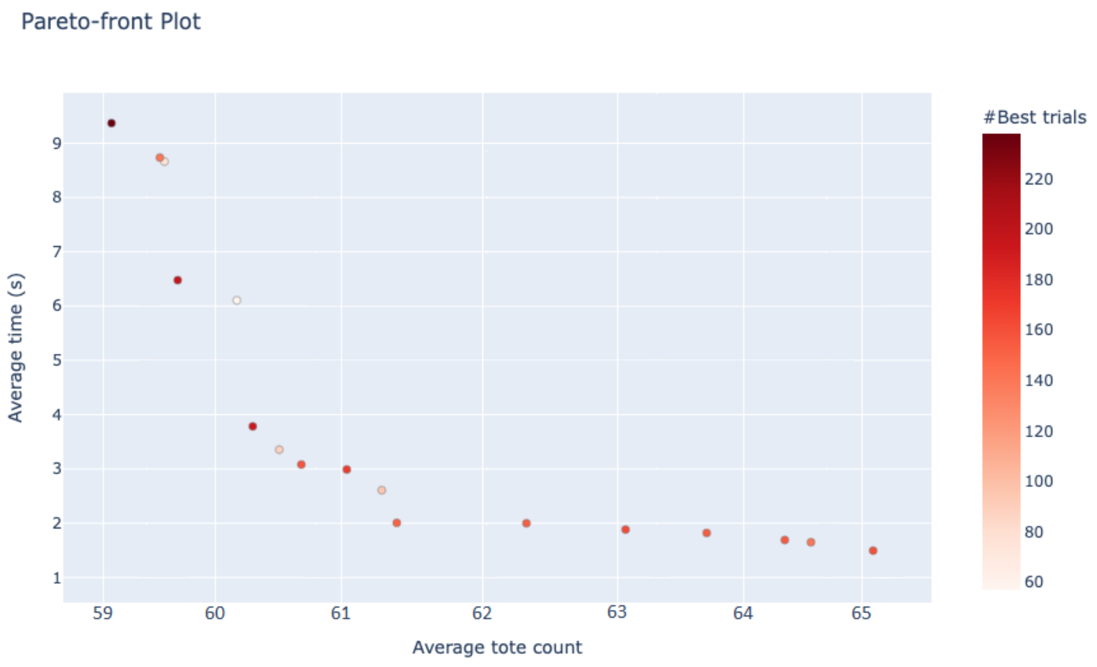


Figure 4.6: The Pareto front of 3D-BAGS' output. Due to confidentiality the average tote count is multiplied with a hidden factor.

As can be seen in Figure 4.6, the algorithm has an average output between 59.10 and 65.26 totes.

While these calculations are done on average between 9.37 and 1.1 seconds per trip respectively. The graph also shows a nice logarithmic shape, which is common for a Pareto front in which both values are minimized.

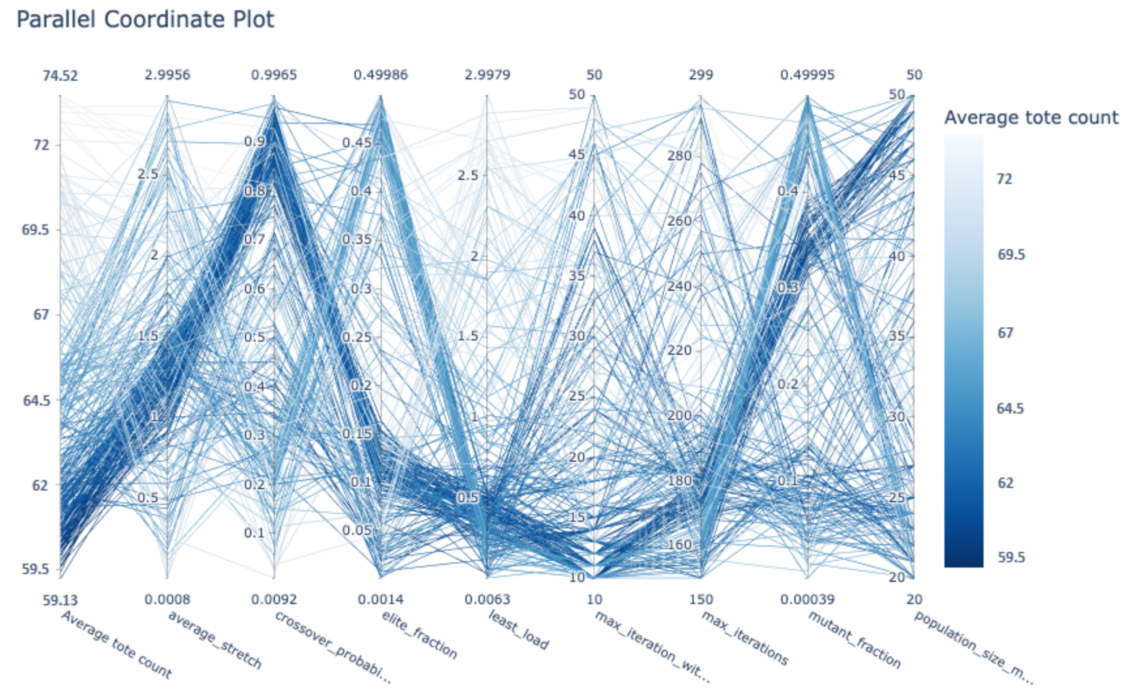


Figure 4.7: The parallel coordinate plot of the average tote count. Due to confidentiality the values for average tote count are multiplied with a hidden factor.

Figure 4.7 shows the connection between different hyperparameters and the average tote count. A line represents a trial and goes from left to right to connect different hyperparameter values. The colour represents the average tote count. The darker the colour, the lower the tote count is. A few interesting observation can be made from this graph.

First of all, a very clear correlation between high crossover probability (third value) and a low average tote count can be observed. This correlation makes sense since having a high crossover probability means that if an 'elite' chromosome is doing crossover with a 'non-elite' chromosome, there is a higher chance that a gene from the 'elite' chromosome is transferred to the child chromosome. This is desirable because one would want a child to inherit most genes of the 'elite', and thus better performing, chromosome. An interesting note is that Gonçalves and Resende found that the crossover probability led to the best results when set to 0.7 [19]. This value was chosen by testing different values from an interval of $[0.70, 0.75, 0.80]$. The graph shows that most good performing trials had a crossover probability higher than this, which could suggest that Gonçalves and Resende could obtain better results if their upperbound was raised.

The elite fraction is shown to lead to the most optimal results when around 0.12. This is mostly in line with the findings of Gonçalves and Resende, who found that a value of 0.10 was the most optimal for the elite fraction. The mutant fraction leads to the best results when set around 0.35, which is much higher than the 0.15 as suggested by Gonçalves and Resende.

The population size multiplier and max iterations have been set by Gonçalves and Resende to 30 and 200 respectively. As can be seen in the graph, the best solutions are found when the population size multiplier is set at around its upper bound (50). This makes sense since having a bigger population (i.e. more chromosomes) leads to more packings to be tried and thus a higher chance of finding a better solution. Surprisingly, max iterations without improvement and max iterations are both almost at their lower bound. For max iterations without improvement it was expected that a higher number would lead to a lower average tote count, since allowing the algorithm to run for longer without improvement might give it more time to find a better solution. Likewise, the expectation was that having a higher value for max iterations would also lead to a lower average tote count because the algorithm would get more iterations to converge to a better solution. Apparently, raising these values does not decrease the average tote count, which could indicate that the algorithm can not converge any further. By looking at at the average fitness value per generation, it shows that it has indeed mostly converged after 150 generations. Average stretch and least load show to be clearly correlated with a low average tote count if both are around 1.3 and 0.3 respectively.

While some hyperparameters have been found to have vastly different optimal values compared to what Gonçalves and Resende found, it is worth noting that their algorithm has been significantly altered for this thesis. It could be that the difference in these hyperparameters are explained by the fact that the values found are more fitted to the Picnic use case compared to the default values. However, it would still be interesting to see whether the best found values for this thesis would also improve their algorithm.

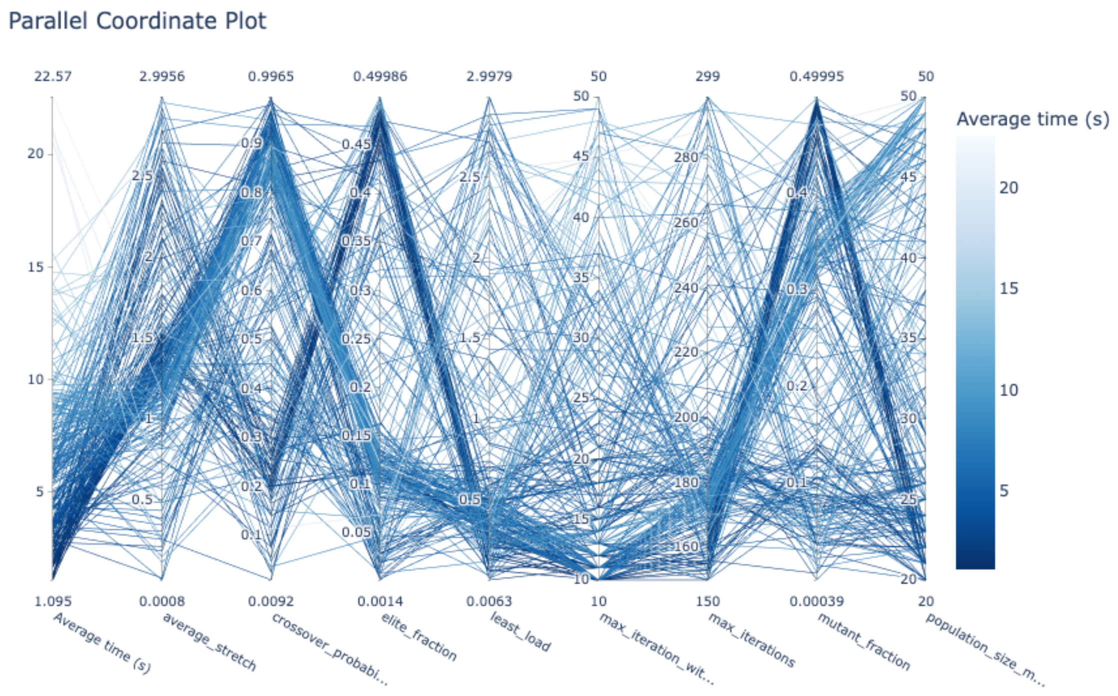


Figure 4.8: The parallel coordinate plot of the average time spent.

Quite a few variables in Figure 4.8 show to lead to a low value for average time spent, while also giving a low value for average tote count. Especially average stretch, least load, max iterations without improvement and max iterations are almost identical to the parallel coordinate plot of average tote count. While this might suggest that these values lead to a low average time spent, the hyperparameter importances calculation indicates that there is actually a different explanation for this.

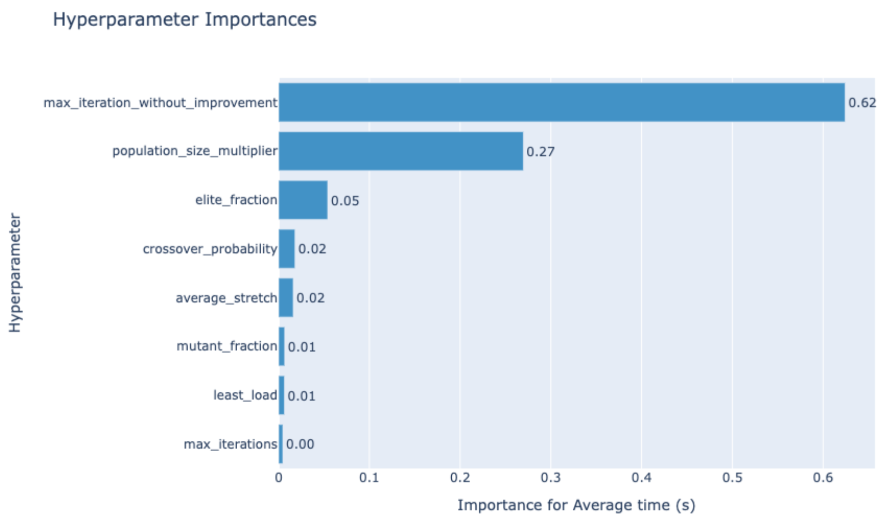


Figure 4.9: Hyperparameter importances for average time spent.

As can be seen in Figure 4.9, some of the aforementioned variables (average stretch, least load and max iterations) all have a very small importance for average time spent in comparison to max iterations without improvement and population size multiplier. These importances have been calculated through FANOVA, which is a combination of random forest models and ANOVA [22].

What is likely to be happening here is that since these hyperparameters do not influence average time spent significantly, the optimizer starts to explore minima for values that do have an effect on average tote count. Figure 4.10 shows that least load and average stretch are indeed the two most important hyperparameters for average tote count.

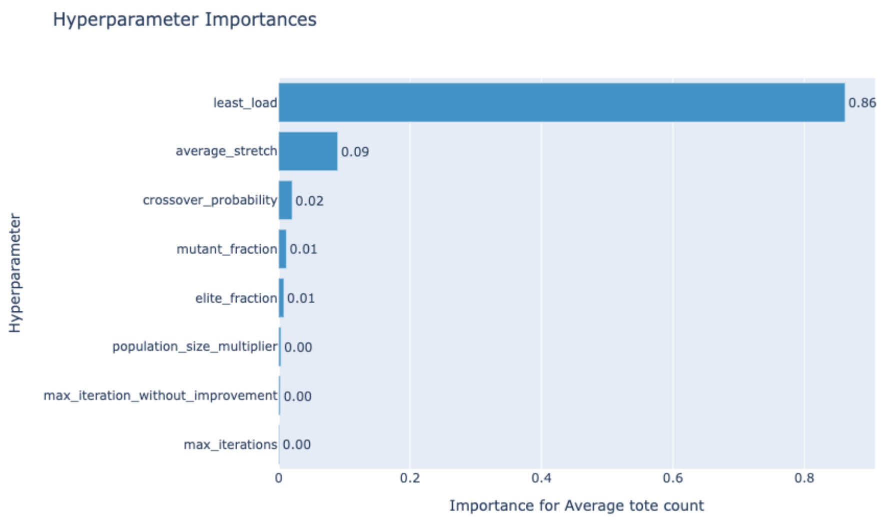


Figure 4.10: Hyperparameter importances for average tote count.

To better explain what is happening an example can be made with least load. The optimizer finds that a low value for least load has a minimizing effect on average tote count and no effect on average time spent. It will thus explore low values for least load and then tweak other hyperparameters that do have an effect on average time spent (e.g. max iteration without increment). Because the optimizer remains at a a low value for least load, it looks as if a low least load value leads to a low amount of average time spent. However, this reduction is actually explained by the values of other hyperparameters.

To conclude, it is observed that max iterations without improvement and population size multiplier are the two most important hyperparameters for average time spent. It was found that a high population size multiplier increases the average time spent but decreases the average tote count. Max iterations without improvement has an negative effect on average time spent and relatively no effect on average tote count, thus the algorithm favours to keep the value of this hyperparameter low.

Least load and average stretch are the two most important hyperparameters for average tote count and both have almost no effect on average time spent compared to other hyperparameters. These two values should be set at around 0.3 and 1.3 respectively. Three configurations for the algorithm are found through the Pareto front in Figure 4.6: quality, balanced and performance. Quality represents the top left configuration on the front and leads to the best packings in the slowest computation time. Balanced is found in the middle of the front and gives worse packings but needs less computation time. Performance is the most bottom right configuration which gives the worst packings but needs the least amount of computation time. These three hyperparameter sets are shown in Table 4.4.

Table 4.4: Hyperparameter configurations for quality, balanced and performance modes. Due to confidentiality the values for average tote count are multiplied with a hidden factor.

Hyperparamer	Configuration		
	Quality	Balanced	Performance
Average stretch	1.08	1.17	1.53
Crossover probability	0.86	0.89	0.23
Elite fraction	0.11	0.10	0.47
Least load	0.24	0.53	0.24
Max iterations without improvement	14	11	10
Max iterations	175	175	160
Mutant fraction	0.34	0.06	0.50
Population size multiplier	50	36	21
Average time spent (s)	9.37	3.78	1.10
Average tote count	59.13	60.24	65.26

4.6. Summary of hyperparameter optimization

By using the stabilization of the coefficient of variation to determine sample size in combination with Multi Objective Tree-structured adaptive Parzen Estimators three hyperparameter configurations were found. The so-called quality configuration leads to the least amount of totes but takes the most time to compute packings with. The quality configuration was found to lead to an average of 59.13 totes, which took on average 9.37 seconds to calculate per trip. The performance configuration can generate packings the fastest, but generates the most totes per trip. This configuration generates on average 60.24 totes in 1.10 seconds per trip. The balanced configuration sits between these two configurations, by generating 65.26 totes in 3.78 seconds on average per trip.

5 Experimental results

This chapter provides the results obtained from multiple experiments that are run on 3D-BAGS. First, the experimental design is described, after which the results of the experiments are reported. Furthermore, some packings that are generated by 3D-BAGS and Picnic's algorithm are performed in real life and compared, which is discussed in Section 5.3.

5.1. Experimental design

The first (and for Picnic most important) experiment, is a between-subjects experiment between Picnic's current approach and 3D-BAGS. This experiment aims to answer the research question: *Does Three-dimensional Bin Packing generate packings with less totes compared to Picnic's current approach?*

In this experiment the independent variable is average tote count (per trip), while the dependent variable is the algorithm that is used. For 3D-BAGS, the quality configuration for the hyperparameters is used, as to show the maximum potential of the algorithm. After some discussion with Picnic, it was decided to compare the algorithms using this configuration because in their experience quite a bit more performance gains are made by more seasoned software developers once they decide whether they want to use an algorithm developed for a thesis. The hypothesis for this experiment is that 3D-BAGS uses less totes than Picnic's current algorithm, because it has a maximum fill rate of 100% instead of 85%.

Besides this first experiment, an ablation study is performed on every new feature that is developed for 3D-BAGS. For each ablation study, hyperparameter optimization is done again, as to plot the new Pareto front for that specific configuration of features. This Pareto front is then compared to the Pareto front in Section 4.5, as to find the influence of the absence of a feature on the algorithm. For these ablation studies, the independent variables are average tote count and average time spent, while the dependent variable is whether the algorithm runs on all features or with a specific feature disabled.

5.1.1. Setup

For the experimental results an Intel Core i7-10700k CPU @ 3.80GHz with 8 cores and 16 GB of memory is used. This is a different setup as used for the hyperparameter optimization in Section 4.1. The reason for this is that the use of a second computer allows for parallelization of hyperparameter optimization and experiments. Since for every ablation study, hyperparameter optimization has to be done again, being able to do other experiments in parallel has been proven to be extremely useful. It should not matter that different hardware is used since only the average tote count is compared upon, which is hardware agnostic. The only exceptions to this are the ablations that compare their Pareto front to the Pareto front found in Section 4.5. To keep these ablations as fair as possible, they are performed on the specifications as described in Section 4.1.

5.2. Comparing to Picnic's algorithm

Picnic provided a data set of trips, which contain deliveries and their corresponding articles. Within this data set it is visible how Picnic's algorithm has divided these deliveries over bags and totes for each trip. To compare 3D-BAGS with Picnic's current approach, 3D-BAGS is run on the same data set after which the difference in the amount of necessary totes can be used as a metric of performance. Unfortunately, it has not been possible to run both algorithms on the same hardware, since the results from Picnic's algorithm were gathered from different hardware to which no access was possible. Because of this, only comparisons can be made on tote count, not on computation time. The comparison is done for each temperature zone, namely ambient, chilled and frozen.

The data provided by Picnic contains 494 trips. Since a confidence interval of 95% with a margin of error of 5% is desired, a sample size of 217 trips needs to be obtained. This sample size is calculated as follows [23]:

$$u = \frac{z^2 * p(1-p)}{\epsilon^2} \tag{5.1}$$

$$sampleSize = \frac{u}{1 + \frac{z^2 * p(1-p)}{\epsilon^2 n}} \tag{5.2}$$

Where n is population size (494), z is the z score (1.96), p is the population proportion (0.5) and ϵ is the margin of error (0.05). Table 5.1 shows the amount of totes needed by 3D-BAGS compared to Picnic's algorithm.

Table 5.1: Tote count for 3D-BAGS and Picnic's algorithm over 217 trips. Due to confidentiality these values are multiplied with a hidden factor.

Algorithm	Temperature zone			
	Ambient	Chilled	Frozen	Total
Picnic	6752	5308	1525	13585
3D-BAGS	6463	5111	1641	13215
Difference	-289 (-4.28%)	-197 (-3.71%)	116 (+7.61%)	-370 (-2.72%)

As can be seen above, 3D-BAGS performs better for ambient and chilled articles, but worse for frozen articles. In total, 3D-BAGS needs less totes than Picnic's current approach. It worth noting that the percentages are not fully in line with the numbers provided in the table. This is caused by rounding errors that occurred when scaling the numbers with a hidden factor. The percentages represent the differences for the original (unscaled) data. The next three sections go over each temperature zone and will explain where these differences come from.

5.2.1. Ambient

The table above shows that out all three temperature zones 3D-BAGS performs the best for ambient articles compared to Picnic's current algorithm. In order to find whether these differences are statistically significant, it is important to first determine whether both algorithms have a normally distributed output. For this the Shapiro-Wilk test can be used, which has the null hypothesis that the data is normally distributed. Since both 3D-BAGS (W(216) = 0.982, P = 0.008) and Picnic's algorithm (W(216) = 0.98, P = 0.004) have a P value that is smaller than 0.05, the null hypothesis is rejected and it can be assumed that the output is not normally distributed. This is further confirmed by visual inspection of the output. Now that it is known whether the output is normally distributed, a decision has to be made whether these two outputs can be regarded as paired or unpaired. Since both algorithms generate a packing that is made from the same input (i.e., specific deliveries which they have to pack), the data is regarded as paired. In other words, every packing of 3D-BAGS is paired to a packing of Picnic's algorithm.

In order to test statistical significance for non-normally distributed and paired data, a Wilcoxon signed rank test can be used. This test has two extra assumptions that need to be met. First of all, the differences per trip need to be independent from each other. Secondly, the distribution of the differences needs to symmetrical. The first assumption is met since the packing of a trip is completely independent of the packing of another trip. The second assumption can be tested by calculating the skewness of the differences [26]. The general consensus is that if the differences have a skewness value of between -0.5 and 0.5, it can be assumed that the differences are approximately symmetrical [25, 1]. For ambient a skewness value of 0.06 was found, which indicates that the data is almost perfectly symmetrical. Thus, the second assumption is also met.

Performing the Wilcoxon signed rank test confirms that there is a significant difference between 3D-BAGS (Mdn = 30) and Picnic's current algorithm (Mdn = 32)[1] on tote count for ambient articles (Z = 1227.0, P < 0.001). Now that it is known whether the differences are significant it is useful to know where these differences occur exactly and what causes them. Figure 5.1 shows the differences of the ambient tote count between the two algorithms.

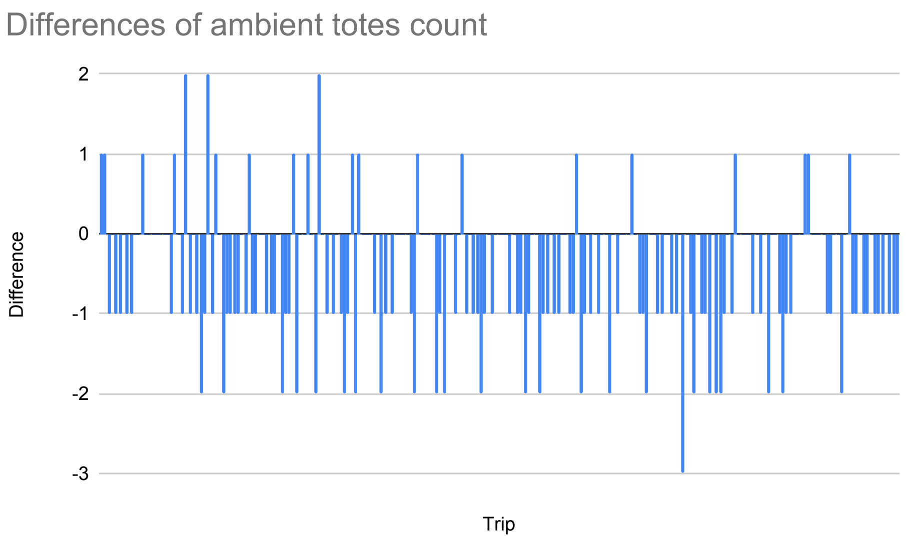


Figure 5.1: The difference of tote count for ambient articles. A negative number means 3D-BAGS outperforms Picnic's algorithm.

What can be seen in this figure is that 3D-BAGS rarely performs worse than Picnic's algorithm. Most of the time both algorithms give the same tote count or Picnic's algorithm outputs an extra tote. For ambient, it was found that only 0.08% of the ambient totes had a fill rate higher than 85%. A three-dimensional model of one of these totes is given in Figure 5.2.

[1]Due to confidentiality these two medians are scaled with a hidden factor.

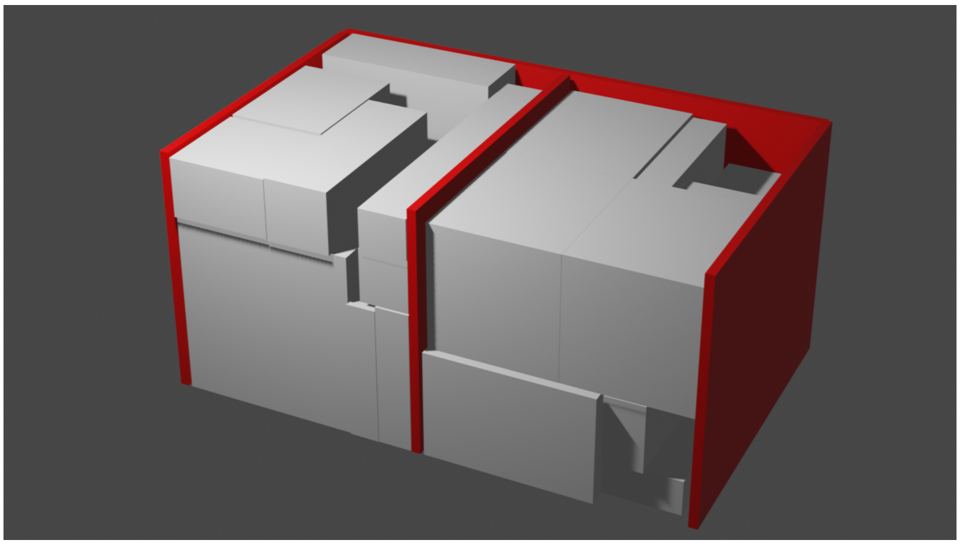


Figure 5.2: An ambient tote with a fill rate above 85%.

It was unexpected that the amount of totes with a fill rate above 85% was so low, especially given that 3D-BAGS still outperforms Picnic's algorithm by 4.28% for ambient packings.

Why Picnic's algorithm uses two less totes in the three outliers, is harder to say. It could be that these packing will cause overflow in real life, but this can not be confirmed. Through 3d modelling it was found that some bags in these outliers have a little bit of overflow, but this could probably be resolved in real life with some minor bag stretching. An example of this overflow is given in Figures 5.3 and 5.4.

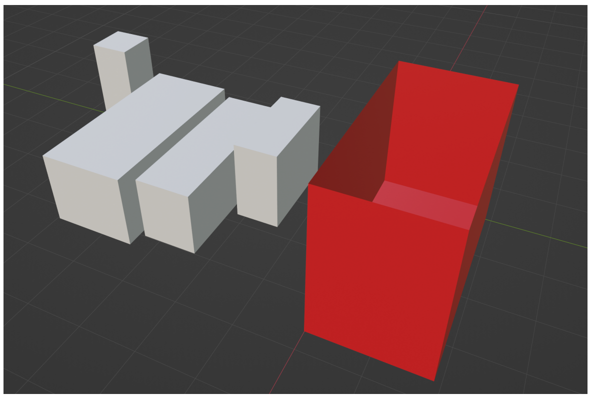


Figure 5.3: The four articles and their corresponding bag.

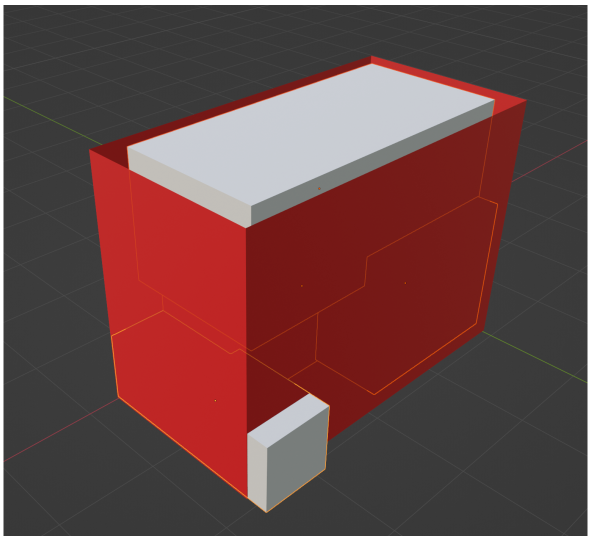


Figure 5.4: The articles packed in the bag.

It could also be that 3D-BAGS would need more generations to converge to a better solution. However, looking at the best fitness of each generation shows that both these outliers most likely have already converged. The most likely explanation is that these outliers contain articles with which 3D-BAGS has some difficulties. For example, both outliers contain 2 packs of kitchen paper. Likewise, one outlier contains 2 packs of toilet paper and the other has 1 pack. Both these articles have been shown through the course of this thesis' development to be more conservatively packed compared to Picnic's algorithm. This is because the algorithm does not consider these articles to be squeezable when placed in a partially filled bag. However, a fulfilment lead has said that these articles are actually often squeezed to be able to fit with other articles.

5.2.2. Chilled

For the chilled output it was found that both 3D-BAGS (W(216) = 0.980, P = 0.003) and Picnic's algorithm (W(216) = 0.978, P = 0.002) reject the null hypothesis of the Shapiro-Wilk test, meaning that it can be assumed that both of their outputs are not normally distributed. A skewness of 0.68 was calculated, which means that the differences for chilled are somewhat asymmetrical. Looking at Figure 5.5, one can see that there is one big outlier that leads to a difference of 4. Removing this outlier reduces the skewnes to 0.28, which makes the distribution somewhat symmetrical. Now, it is undesirable to remove a data point just to meet an assumption of the Wilcoxon signed rank test. A better approach would be to transform the data such that it becomes less skewed. To unskew the data, it would make sense to transform the data such $f(x) = x^{\frac{1}{3}}$. Because the Wilcoxon signed rank test analyses the rank of the data and the aforementioned transformation preserves this rank, the transformed data can be used for statistical testing. The transformation leads to a skewness value of -0.33 which indicates that it is now approximately symmetrical, thus the Wilcoxon signed rank test can be performed. The Wilcoxon signed rank test shows that there is a significant difference in tote count for chilled articles between 3D-BAGS (mdn = 24) and Picnic's algorithm (mdn = 24)[2] (Z = 1823.0, P < 0.001).

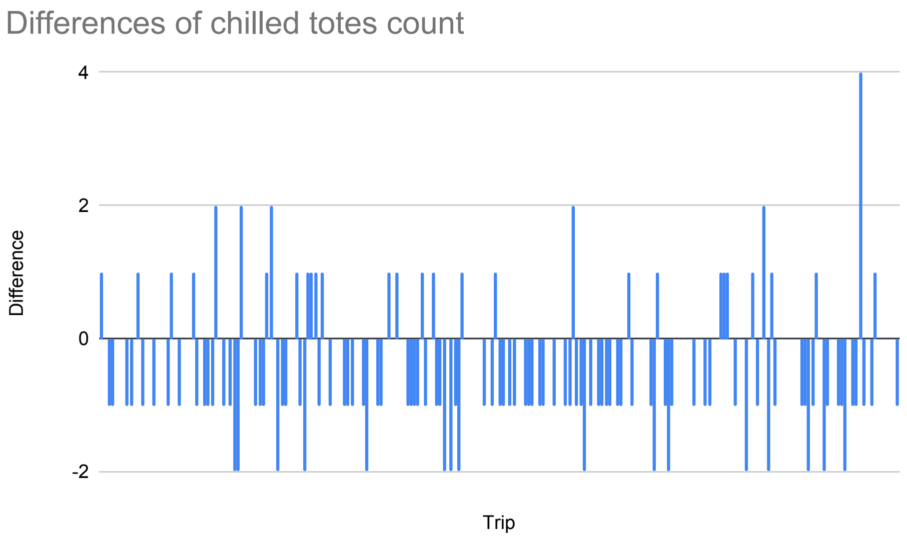


Figure 5.5: The difference of tote count for chilled articles. A negative number means 3D-BAGS outperforms Picnic's algorithm.

As can be seen above, the graph of differences is somewhat similar to the graph of differences for ambient totes. Most packings have an equal tote count for both algorithms or 3D-BAGS uses one less tote. A few times Picnic's algorithm performs better and uses one or even rarely two totes less. Likewise, 3D-BAGS also has a few packings that use two less totes. Most notably is the outlier for which Picnic's algorithm uses four totes less.

It was found that 0.74% of the chilled totes had a fill rate higher than 85%. This is a higher percentage than for ambient totes, which was not expected since the reduction of tote count (in percentages) is higher for ambient than for chilled.

[2]Due to confidentiality these two medians are scaled with a hidden factor.

5.2.3. Frozen

Similarly to ambient and chilled packings, the Shapiro-Wilk test also shows that both 3D-BAGS (W(216) = 0.897, P < 0.001) and Picnic's algorithm (W(216) = 0.884, P < 0.001) are not normally distributed for the frozen output. A skewness of 0.74 was found for the differences. This is, similarly to chilled, caused by an outlier. Transforming the data such that $f(x) = x^{\frac{1}{3}}$ leads to an skewness of 0.05, which indicates that the differences are approximately symmetrical after transformation.

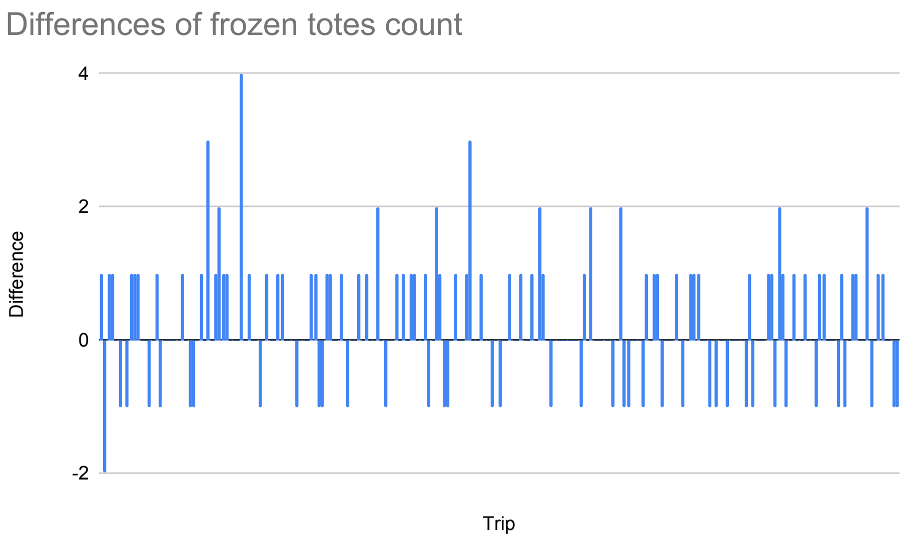


Figure 5.6: The difference of tote count for frozen articles. A negative number means 3D-BAGS outperforms Picnic's algorithm.

The Wilcoxon signed rank test shows that no significant difference can be assumed between 3D-BAGS (mdn = 8) and Picnic's algorithm (mdn = 8)[3] given their tote count for frozen articles (Z = 2484.5, P = 0.20). However, there is something important that should be mentioned which could influence the results of this test. As can be seen in Figure 5.6, frozen has a lot more packings that have an equal tote count compared to ambient and chilled. This might be a problem because, as suggested by Wilcoxon himself, if the difference between a pair is zero, it should be omitted [40]. Since so many data points are omitted, one could argue that the effective sample size is too small for the Wilcoxon signed rank test to be reliably executed. Fortunately, a method was developed by Pratt which can deal with these zeros [35]. Performing the Wilcoxon signed rank test given Pratt's method of including zeros shows that the differences for frozen totes are statistically significant (Z = 6774.5, P = 0.02).

Out of all the frozen totes generated by 3D-BAGS, not a single tote had a fill rate higher than 85%. This makes sense since customers often do not order a lot of frozen articles and thus bags are almost never tightly packed. Picnic's current algorithm has a relatively low mean bag fill rate for frozen totes[4], which makes it clear that surpassing 85% for an entire tote is very hard to do and thus only happens rarely or, in the case of the data provided by Picnic, not at all. Even though the max fill rate of 85% is never surpassed, it is still surprising that 3D-BAGS performs substantially worse than Picnic's current algorithm. The reason for this increase in average tote count is explained in the next section.

5.3. Real life comparison

Two big outliers for frozen deliveries have been compared in real life. The first comparison is made for a trip where 3D-BAGS performs better than Picnic and the second comparison is made for a trip

[3]Due to confidentiality these two medians are scaled with a hidden factor.

[4]The actual mean fill rate for frozen totes is given in the confidential version of this thesis.

where the opposite is true. Only frozen deliveries are compared in real life because of two reasons. Firstly, while it can be speculated that the algorithm performs better than Picnic for ambient and chilled because of the increased max fill rate or bag stretching, it was unknown why the algorithm falls short for frozen articles. Secondly, there was only a limited amount of time that could be spent in a fulfillment centre to perform the packings. In order to make fair comparisons, entire trips containing multiple deliveries need to be packed for both algorithms. Unfortunately, ambient and chilled on average need much more totes in a trip than frozen[5]. It is impossible to test that many totes for ambient and chilled within the time span that was available.

5.3.1. Picnic performing better than 3D-BAGS

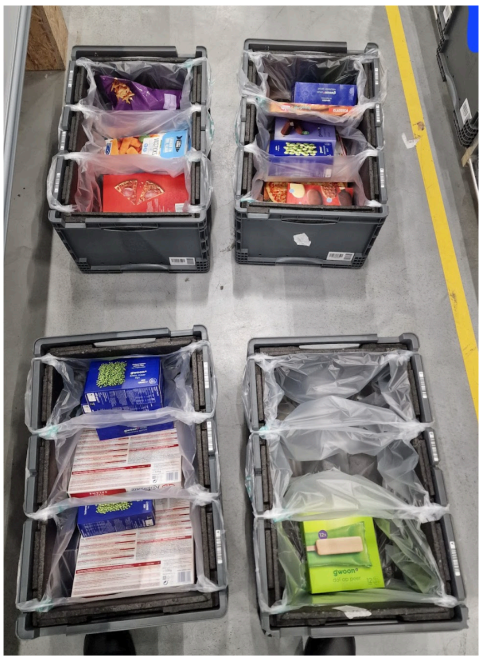


Figure 5.7: The packing of Picnic's algorithm.

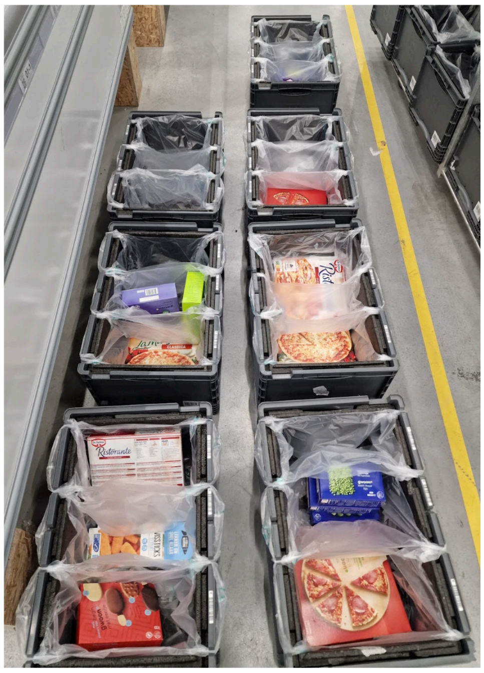


Figure 5.8: The packing of 3D-BAGS.

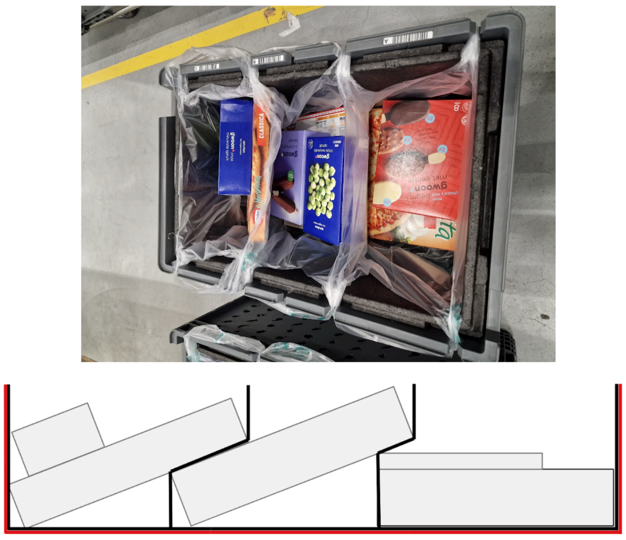


Figure 5.9: A packing made by Picnic's algorithm in which bags overlap.

[5]The actual mean values per temperature zone are provided in the confidential version of the thesis.

Figures 5.7 and 5.8 show a packing in which Picnic's algorithm needs three totes less. What can be seen in Figure 5.9 is that the solution given by Picnic's algorithm only fits if articles are overlapping over each other between different bags. While in real life this is possible, 3D-BAGS does not allow for this. Not allowing bags to overlap is a notable downside for especially frozen totes because this temperature zone has a lot of articles that only fit together if their bags can overlap (e.g., pizza boxes).

5.3.2. 3D-BAGS performing better than Picnic

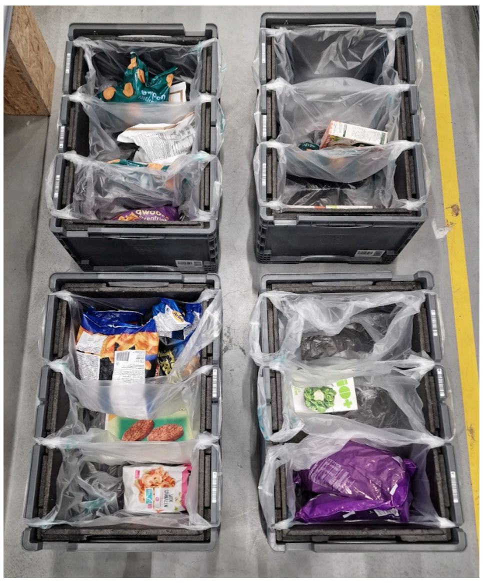


Figure 5.10: The packing generated by Picnic's algorithm.

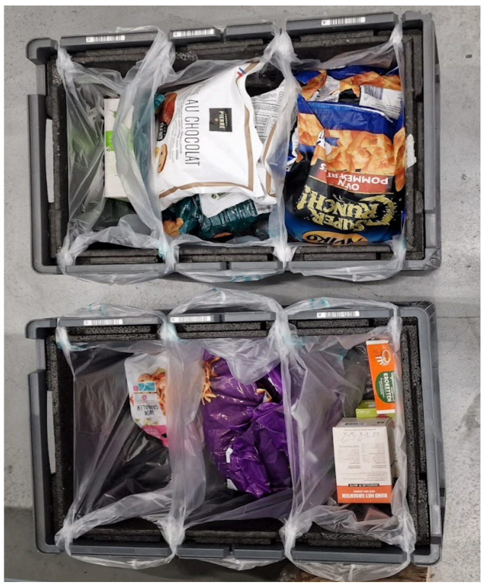


Figure 5.11: The packing generated by the 3D-BAGS algorithm.

As shown in Figures 5.10 and 5.11, the solution given by Picnic requires two more totes. One can see that compared to the solution given by 3D-BAGS, the bags are not as compactly packed. This happens because Picnic's algorithm does not allow for bag stretching. Take the middle bag in the upper tote in Figure 5.11. This bag is filled with 10.6 liters by 3D-BAGS. According to Picnic's algorithm, a bag is $\frac{1}{3}$ of a tote, which is 9.1 liters for frozen totes. Thus, Picnic's algorithm divides this bag into two bags. However, as a result of this, there now two bags the size of $\frac{1}{3}$ of a tote which are both filled to almost a half of their total volume. By allowing for bag stretching, 3D-BAGS uses bigger bags, but also fills them more compactly. These bigger bags can then be combined with bags that are negatively stretched, as can be seen in the top left bag of Figure 5.11. To summarise, the inclusion of bag stretching has allowed 3D-BAGS to pack totes more compactly, by combining positively and negatively stretched bags.

5.4. Ablation study

The following section contains experiments in which a specific feature is disabled. By comparing the average tote count and/or the average time spent with and without a feature, a conclusion can be drawn whether its implementation is beneficial for 3D-BAGS. Because 3D-BAGS is an extension of the original BRKGA, the effect of each feature found for 3D-BAGS also gives some indication of what its effect would be on BRKGA. Thus, these ablation studies could also be used to show that certain features would be good (or bad) additions to BRKGA.

5.4.1. Bag stretching

By not allowing bag stretching, three things happen. First of all, more articles need to be squeezed in order to fit inside of an empty bag. The amount of squeezed articles is increased roughly four times if no bag stretching is allowed. The exact numbers can be found in the confidential version of this thesis. Note that this is not the frequency of squeezing, but the amount of distinct articles that are squeezed. This is huge increase and could lead to a more inaccurate model if these articles are not squeezable in real life. Secondly, disabling bag stretching can lead to more damaged goods since more unsqueezable articles are squeezed in order to fit in an empty bag. Thirdly, bags that are almost empty still take up space the size of $\frac{1}{3}$ of a tote if they can not be stretched negatively (i.e., shrinked). This leads to less tightly packed totes. In Figure 5.12 the Pareto front of 3D-BAGS with and without bag stretching is compared.

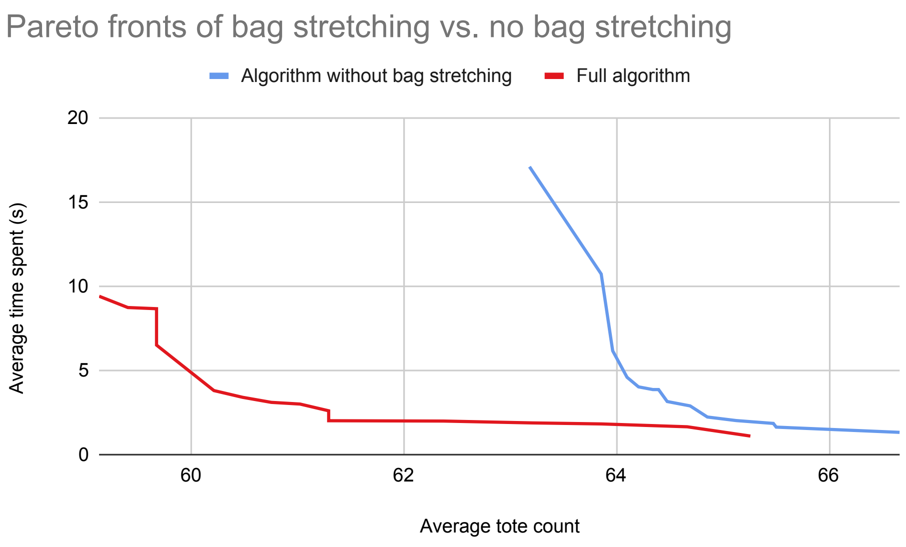


Figure 5.12: The Pareto front of 3D-BAGS without bag stretching compared to 3D-BAGS with bag stretching. Note that due confidentiality the average tote count is multiplied with a hidden factor.

This figure shows that while the average time spent is somewhat the same, there is a big difference in average tote count. Based on the increase of average tote count and amount of squeezed articles, it is safe to say that the inclusion of bag stretching has a positive influence on the packing quality provided by the algorithm.

5.4.2. Squeezability

The squeezability property was introduced to allow for tighter packings when articles behave almost fluid-like. The expectation is that by allowing squeezable articles to be be squeezed inside partially filled bags, no new bag has to be opened and thus less totes are needed to ship the groceries.

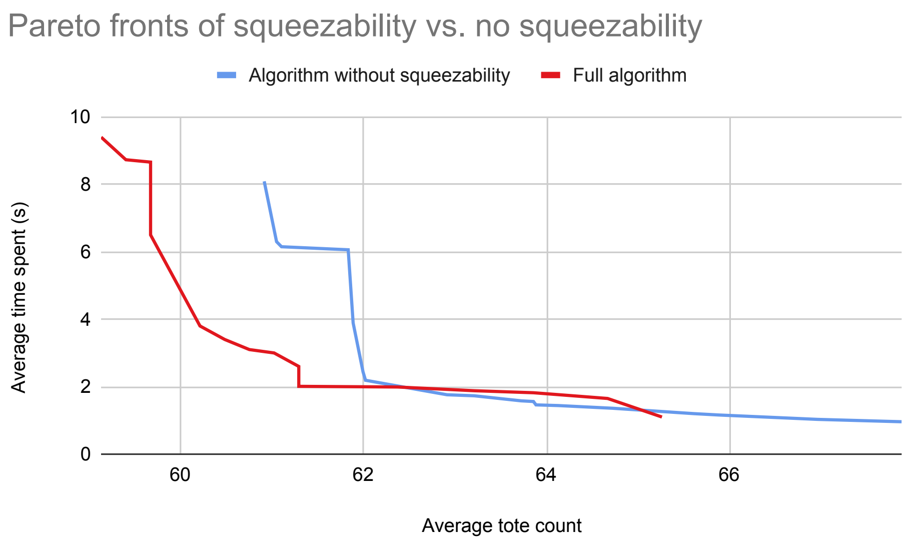


Figure 5.13: The Pareto front of 3D-BAGS without the squeezability property compared to 3D-BAGS with the squeezability property.

What can be seen in Figure 5.13 is that the inclusion of the squeezability property, which was conservatively decided for only the 100 most bought articles per temperature zone, already has a significant effect on average tote count. The squeezability property has less of an effect for packings with a higher average tote count because these packings are less compact and articles are thus not required to be squeezed to fit. Average time spent does not look to be influenced by this property, which was anticipated beforehand.

5.4.3. Multithreading

Unfortunately, disabling multithreading leads to such an increase in computation time, that generating a Pareto front is not feasable within the timeframe of this thesis. However, a comparison can still be made with multithreading disabled, given the hyperparameters as found in Section 4. A sample size of 217 is used, as calculated in Section 5.2.

There are two multithreading aspects in the algorithm. Each delivery within a trip is processed in parallel and chromomsomes within a delivery are generated and evaluated at the same time. Ideally, the ablation for both of of these multithreadings should be researched. However, disabling multithreading per delivery would free up resources to generate and evaluate more chromosomes in parallel, so the effect of ablation would be an unrealistic measurement. One could argue that it would be fair to split the cores of a computer into two and assign the multithreading per delivery to one half of the cores and the multithreading per chromosome to the other half. However, this would eliminate the effect of prioritized multithreading which is explained in Section 3.3.1. Because of the reasons above it was determined that the most fair comparison could be done by either allowing full multithreading or no multithreading at all.

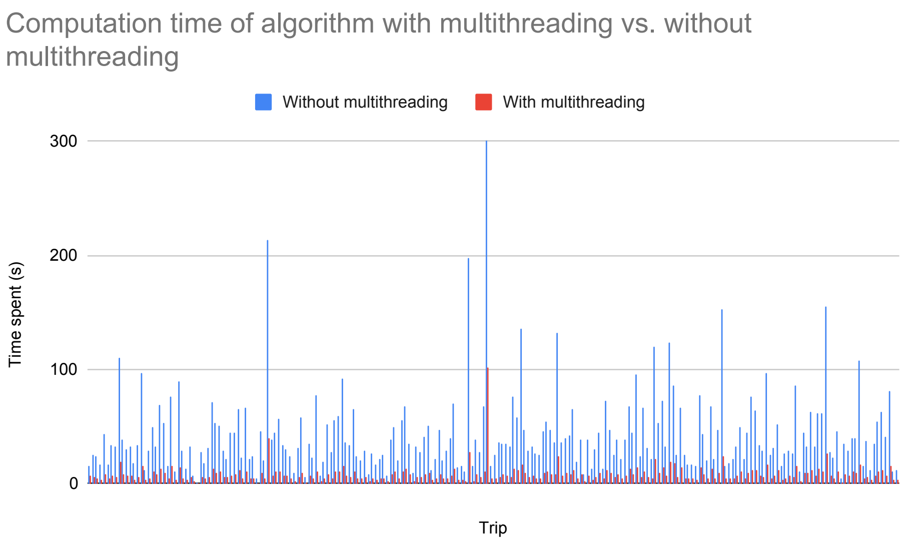


Figure 5.14: The computation time of the algorithm with multithreading vs. without multithreading. Note that the bar that ends at 300 s actually has a value of 800, but this is not within the range of the graph (for readability).

Figure 5.14 shows that enabling multithreading has an immense effect on the reduction of the time spent per trip. Without multithreading, a trip takes on average 45.0 seconds to process. However, if multithreading is enabled, this number is reduced to 8.4 seconds. In other words, the usage of multithreading leads to a decrease of 81.33% in computation time when using an i7-10700k processor. This decrease will vary depending on the machine and the amount of cores used.

5.4.4. Memoization of packings

Since 3D-BAGS uses elitism, a certain fraction of the chromosomes are copied over to the next generation. Because of this, the expectation is that the memoization of packings would lead to a performance increase since they would not need to be recalculated. Because we want to compare the effectiveness of memoization, it was decided to leave the generation of the key for the caching of the chromosomes enabled for both the full algorithm and the algorithm with memoization disabled. It could be that a better key generation method could be found which would lower the average time spent for both variants. Below, the Pareto front of 3D-BAGS with no memoization of packings is compared to the full algorithm.

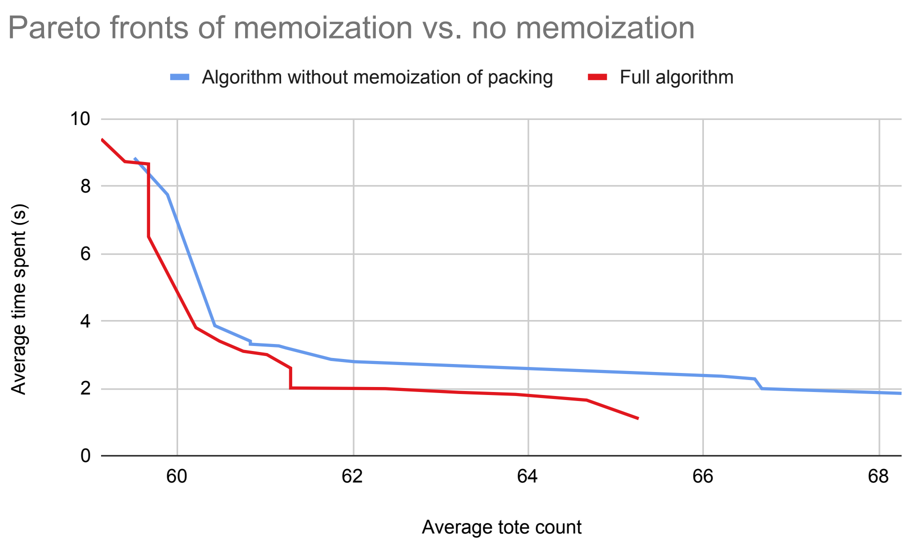


Figure 5.15: The Pareto front of 3D-BAGS without memoization of packings compared to 3D-BAGS with memoization of packings.

What can be seen in Figure 5.15 is that there is a slight decrease in computation time when memoization is enabled. However, one can also see a decrease in average tote count for the full algorithm. In theory this should not happen since memoization returns the exact same packing as the algorithm if no memoization is used, given the same chromosome. This small difference is most likely explained by the stochastic nature of 3D-BAGS. 3D-BAGS is stochastic because for every run different chromosomes are generated, which can lead to a different convergence per delivery. However, one could argue that average time spent is less affected by this stochasticity, since the amount of chromosomes that need to be packed remains unchanged. Still, average time spent is not completely deterministic. The amount of returns from the cache can be different, depending on how many equal packings are generated.

Because of the reasons mentioned above, no definitive conclusion can be drawn on the performance impact of the memoization of packings. It appears as if there is a slight decrease in computation time for low average tote counts and a decrease of around 40% for hyperparameter sets in which the average time spent is already quite low. All in all, the data seems to support the hypothesis that the inclusion of memoization has a positive, albeit relatively small, effect on the performance of 3D-BAGS.

5.4.5. Memoization of diagonal rotations

It has been observed that the memoization of diagonal rotations is such an integral part of 3D-BAGS, that disabling it grinds the algorithm to a halt. It would take an immense amount of computation power to do a proper ablation study for this feature, which is unfortunately not possible for this thesis. However, a rough estimation of the impact of memoization of diagonal rotations can be made. This estimation is done as follows. For each article that is diagonally rotated, the time it took to find a solution is obtained. This time is multiplied by the amount of times this article is placed by 3D-BAGS. The resulting number is the total time for that specific article that is lost when memoization is not applied. By summing this value for each article and dividing it by the amount of trips, the average time spent on the calculation of diagonally rotated articles is found. Adding this value to the average time spent of the full algorithm allows a rough comparison to be made. The following values were found for 58 trips.

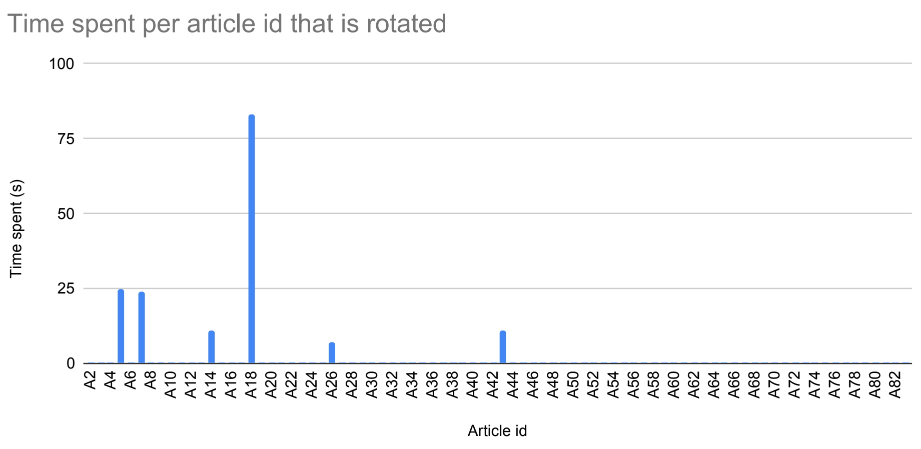


Figure 5.16: The time spent per article id that is diagonally rotated. Due to confidentiality the article id is hidden.

As can be seen in Figure 5.16, most articles need almost no time to find a solution, these are the articles that are rotated once through the simple pythagoras formula as described in 3.1.6 or the articles in which the solver quickly finds that no solution exists (i.e. it can not fit without squeezing). The articles that do have a large value for average time spent are the articles that have a double diagonal rotation. Doing the estimation as described before, it was found that it would take roughly 3.7×10^8 seconds to calculate the double diagonal rotation for these articles on an i7-10700k processor. This number is so extremely high because each of these articles is placed between 2.6×10^4 and 4.19×10^6 times. This makes sense since the Biased Random-Key Genetic Algorithm will try a lot of chromosomes in which these articles have to be placed. If no memoization is done on diagonal rotations, a trip would take on average 6×10^6 seconds to be processed, which is an increase of 3.6×10^7% compared to the average time spent of 17.5 seconds for the full algorithm (which includes the calculation of diagonally rotated articles once). The immense increase in percentage makes it clear that 3D-BAGS can not function normally without the memoization of diagonal rotations.

5.5. Summary of obtained results

To summarise, 3D-BAGS outperforms Picnic's current approach for ambient and chilled totes, but is considerably outperformed for frozen packings. All these results have been shown to be statistically significant. 3D-BAGS leads to suboptimal packings for frozen articles because it does not allow bags to overlap. The inclusion of bag stretching and a squeezability property has been shown to lead to a decrease in tote count. Furthermore, the use of multithreading and memoization leads to a decrease in computation time.

6 Discussion & conclusion

This thesis expands upon the state-of-the-art Three-dimensional Bin Packing algorithm BRKGA by including bag stretching, packing zones, diagonal rotations and squeezing. Furthermore, multiple performance improvements have been made through the implementation of memoization, priority based multithreading and a different stopping criterion. These improvements have been applied to a two-stage algorithm called 3D-BAGS, which can pack articles into bags and distribute these bags into totes. 3D-BAGS has been proven to generate packings that require less totes compared to the algorithm Picnic currently uses. Besides this, 3D-BAGS reduces the amount of overflow compared to Picnic's current algorithm by taking the width, height, length and exact placement of each article into account.

This chapter first discusses the characteristics of 3D-BAGS that lead to the results obtained in Chapter 5. After this, a conclusion is drawn and the research questions stated in Chapter 1 are answered. Based on the results, a recommendation is made for Picnic and future work is proposed.

6.1. Discussion

While a large part of the results have already been interpreted in Chapter 5, there are still some advantages and (possible) shortcomings of 3D-BAGS that should be discussed. This section serves to underline these characteristics, which were not mentioned in the previous chapter. Other limitations (i.e., features that are not implemented) of 3D-BAGS are discussed in Section 6.4.

6.1.1. Bag stretching

As shown in Chapter 5, 3D-BAGS outperforms Picnic's current algorithm for ambient and chilled totes, but has a higher number of frozen totes for the same trips. In the end of Chapter 1 it is hypothesized that the use of Three-dimensional Bin Packing would allow for a higher maximum fill rate and thus less totes. However, Chapter 5 shows that 3D-BAGS rarely surpasses the maximum fill rate of 85% that is used by Picnic's current algorithm. Thus, other characteristics of 3D-BAGS must cause this decrease in tote count. The ablation study for bag stretching showed that this feature has a significant effect on the average tote count. During experimentation many of instances were also found in which bag stretching was the main cause for a reduction in tote count. The explanation for this is that in Picnic's algorithm a bag is generally regarded as $\frac{1}{3}$ the volume of a tote. Thus, if a customer's order needs only slightly more than this, the algorithm will open up a whole new bag that is again $\frac{1}{3}$ of a tote. This can lead to a significantly large amount of unused space, as is shown in Figure 6.1.

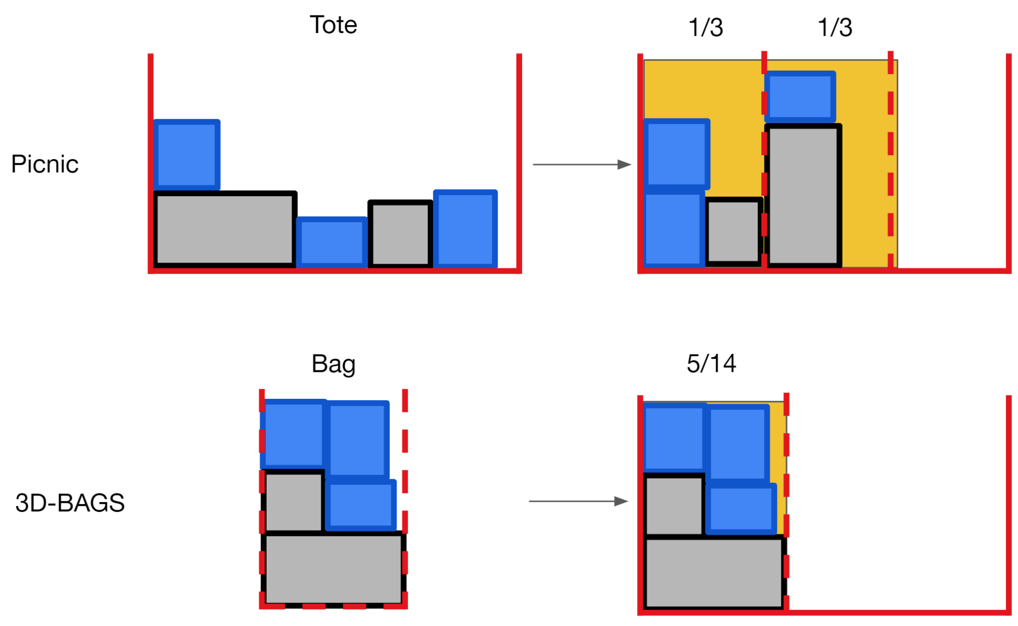


Figure 6.1: The effect of bag stretch on fill rate, where yellow represents the unused space in a bag. Picnic's algorithm starts by assigning articles to totes and then splitting these totes into bags, while 3D-BAGS first generates the bags after which it distributes them over totes.

By including bag stretching, more compact packings can be generated, which leads to a decrease in tote count. The same logic applies when a customer orders less than one bag. By negatively stretching (shrinking) this bag, other bags in the same tote have more space to stretch out and thus more articles can be packed in the same bag, which (often) leads to less totes. It is worth noting that bag stretching can not be incorporated into Picnic's current One-dimensional Bin Packing algorithm, since the algorithm should know the articles' dimensions and placement locations in order to determine the amount of stretch per bag.

6.1.2. Maximum fill rate

3D-BAGS was run under the assumption that a maximum fill rate of 100% could be used. Picnic's current algorithm uses a maximum fill rate of 85% because it can not guarantee that no overflow occurs. Even though 3D-BAGS generates packings in which no overflow occurs, the use of a maximum fill rate of 100% might impose some problems operationally. This is because (human) pickers might not pack bags exactly as computed by 3D-BAGS, which could lead to inefficient packings and thus overflow. While it is possible to instruct pickers where to exactly place articles, this is not something Picnic is currently planning to do. Thus, it might be necessary to lower the maximum fill rate as to prevent overflow. The hypothesis is that this will not have a significant effect on the performance of 3D-BAGS, since totes already rarely surpass 85%.

6.1.3. Squeezed oversized articles

One significant advantage of 3D-BAGS is that the use of Three-dimensional Bin Packing can prevent overflow if the shape of an article does not fit the assigned unused volume. However, due to the product range Picnic offers, 3D-BAGS is (rarely) forced to squeeze articles in order to fit into an empty, but fully stretched, bag. While most of the articles that are squeezed can be squeezed in real life, as shown in Appendix C, there are some articles that might not be squeezable. Thus, there might be some rare instances in which overflow still occurs because an unsqueezable article is squeezed by 3D-BAGS.

6.2. Conclusion

Based on the results and discussion, the research questions stated in Chapter 1 can be answered as follows:

1. **What Three-dimensional Bin Packing algorithm could be used to efficiently pack groceries into totes and how should this algorithm be expanded upon to fit the Picnic use case?**
   A Biased Random-Key Genetic Algorithm can be used in combination with Empty Maximum Spaces and the Distance to the Front-Top-Right Corner heuristic to effectively pack groceries into bags. These bags are can then be packed into totes by again using a Biased Random-Key Genetic Algorithm and a first fit heuristic. The algorithm needs be expanded with features that allow for single or double diagonal rotation, as to be able to include articles that have one or more dimensions that are bigger than a bag. Bag stretching should also be modelled, since a substantial amount of articles can not fit into $\frac{1}{3}$ of a tote, even if they are diagonally rotated. Furthermore, the algorithm should allow articles to be squeezed if they behave fluid-like or do not fit into a fully stretched bag. Lastly, the algorithm should be able to deal with fragility by incorporating the packing zones during gene sorting when generating packing chromosomes.

   For more practical reasons, the algorithm should include multithreading, memoization and a stopping criterion (maximum amount of generations, maximum of generations without improvement and time limit) as to reduce the computation time and make it run in an acceptable time frame given Picnic's use case.
2. **Can Three-dimensional Bin Packing lead to a fill rate higher than 85% given Picnic's product range?**
   Ambient and chilled totes have been shown to reach a fill rate above 85%. However, this only happens for 0.08% and 0.74% of the generated totes respectively. For frozen, not a single tote has been generated with a fill rate higher than 85%.
3. **Does Three-dimensional Bin Packing generate packings with less totes compared to Picnic's current approach?**
   By using 3D-BAGS, as described in Chapter 3, a reduction of 4.28% has been found for ambient totes. Chilled totes could also be reduced by 3.71%. The amount of frozen totes have been shown to increase by 7.61%. In total 3D-BAGS uses 2.72% less totes than Picnic's current approach, given the data provided by Picnic.
4. **How can a Three-dimensional Bin Packing deal with articles that have one or more dimensions that are bigger than the bag in which they need to be placed?**
   By using Mixed Integer Quadratic Programming, the single or double diagonal rotation of articles can be calculated, which allows articles to fit even if one or more dimensions are bigger than the bag. Furthermore, by incorporating bag stretching, a bag can fit such oversized articles by stretching up to half a tote. As a last resort, by incorporating the squeezing of an article it can be made to fit into a fully stretched bag if it does not fit with a single or double diagonal rotation.

6.3. Recommendation to Picnic

The recommendation can be found in the confidential version of the thesis.

6.4. Future work

While 3D-BAGS is a more accurate model of reality compared to Picnic's current approach, there are some more features that could be implemented as to make its packings even more realistic. These features are discussed below.

- **Irregular items**
  Currently, Picnic only knows the width, height and length of an article. In reality, however,

articles are not always cuboids. To obtain an even more accurate packing model one could take the precise shape of these so called irregular items into account when generating a feasible packing. Recent research done by Wang and Hauser has shown that the packing of irregular items can be done in an online environment [39]. It would be interesting to see whether this can also be done in an offline environment and more specifically as an extension to BRKGA or 3D-BAGS.

- **Stability**
  While initially in scope of this thesis, stability has been dropped from 3D-BAGS as Picnic did not think it would be a necessary addition (since pickers do not follow the exact packing given by the algorithm). However, it would still be interesting to see how the inclusion of stability would influence tote count. If the effect is minimal it would only make the packings generated by 3D-BAGS more realistic. As shown in Chapter 2, vertical stability could easily be incorporated by limiting the width and length of an EMS by the width and length of the articles below it. However, it might also be good to find methods that allow for stability in which only parts of an article are supported on the bottom, as is done with the top down matrix placement heuristic discusses in Chapter 2.
- **Online variant**
  As of recently, Picnic has opened its first automatic fulfilment centre (FCA). Due to confidentiality, the exact workings of FCA are only outlined in the confidential version of the thesis. The conclusion is that an offline algorithm is not suitable for this kind of a fulfilment centre. In order to have 3D-BAGS work for such use case, one could look into ways to transform it into an online algorithm.

As a final remark, I would suggest future work on Three-dimensional Bin Packing algorithms to not use the data generation developed by Martello et al. [31] and Berkey and Wang [6]. As mentioned by Bortfeldt and Wäscher, these data sets are not as challenging anymore and do not fully represent the constraints found in real life (e.g., it does not include oversized articles as found in Picnic's data set) [8]. Based on the literature review done in Chapter 2, I believe this research area has matured enough to shift over to a more complex and thus challenging data set, which will allow for comparisons that are more inline with real world constraints.

References

[1] URL: https://docs.oracle.com/cd/E57185_01/CBREG/ch03s02s03s01.html.

[2] *A hyperparameter optimization framework*. URL: https://optuna.org/.

[3] M. Alicastro. *heuristics*. https://github.com/mirkoalicastro/heuristics. 2019.

[4] I. Araya and M.-C. Riff. "A beam search approach to the container loading problem". In: *Computers Operations Research* 43 (2014), pp. 100–107. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2013.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0305054813002530.

[5] J. Bergstra et al. "Algorithms for Hyper-Parameter Optimization". In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor et al. Vol. 24. Curran Associates, Inc., 2011. URL: https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf.

[6] J. O. Berkey and P. Y. Wang. "Two-Dimensional Finite Bin-Packing Algorithms". In: *The Journal of the Operational Research Society* 38.5 (1987), pp. 423–429. ISSN: 01605682, 14769360. URL: http://www.jstor.org/stable/2582731.

[7] E.E. Bischoff and M.S.W. Ratcliff. "Issues in the development of approaches to container loading". In: *Omega* 23.4 (1995), pp. 377–390. ISSN: 0305-0483. DOI: https://doi.org/10.1016/0305-0483(95)00015-G. URL: https://www.sciencedirect.com/science/article/pii/030504839500015G.

[8] A. Bortfeldt and G. Wäscher. "Constraints in container loading – A state-of-the-art review". In: *European Journal of Operational Research* 229.1 (2013), pp. 1–20. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2012.12.006. URL: https://www.sciencedirect.com/science/article/pii/S037722171200937X.

[9] M. Boschetti. "New lower bounds for the three-dimensional finite bin packing problem". In: *Discrete Applied Mathematics* 140 (May 2004), pp. 241–258. DOI: 10.1016/j.dam.2003.08.004.

[10] C.S. Chen, S.M. Lee, and Q.S. Shen. "An analytical model for the container loading problem". In: *European Journal of Operational Research* 80.1 (1995), pp. 68–76. ISSN: 0377-2217. DOI: https://doi.org/10.1016/0377-2217(94)00002-T. URL: https://www.sciencedirect.com/science/article/pii/037722179400002T.

[11] T. G. Crainic, G. Perboli, and R. Tadei. "Extreme Point-Based Heuristics for Three-Dimensional Bin Packing". In: *INFORMS Journal on Computing* 20 (Aug. 2008), pp. 368–384. DOI: 10.1287/ijoc.1070.0250.

[12] G. B. Dantzig and P. Wolfe. "Decomposition Principle for Linear Programs". In: *Oper. Res.* 8.1 (Feb. 1960), pp. 101–111. ISSN: 0030-364X. DOI: 10.1287/opre.8.1.101. URL: https://doi.org/10.1287/opre.8.1.101.

[13] A. P. Davies and E. E. Bischoff. "Weight distribution considerations in container loading". In: *European Journal of Operational Research* 114.3 (1999), pp. 509–527. ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(98)00139-8. URL: https://www.sciencedirect.com/science/article/pii/S0377221798001398.

[14] K. Deb et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017.

[15] S. P. Fekete, J. Schepers, and J. C. van der Veen. *An exact algorithm for higher-dimensional orthogonal packing*. 2006. DOI: 10.48550/ARXIV.CS/0604045. URL: https://arxiv.org/abs/cs/0604045.

[16] X. Feng, I. Moon, and J. Shin. “Hybrid genetic algorithms for the three-dimensional multiple container packing problem”. In: *Flexible Services and Manufacturing Journal* 27.2-3 (2013), pp. 451–477. DOI: 10.1007/s10696-013-9181-8.

[17] P. I. Frazier. *A Tutorial on Bayesian Optimization*. 2018. DOI: 10.48550/ARXIV.1807.02811. URL: https://arxiv.org/abs/1807.02811.

[18] J. F. Gonçalves and R de Almeida J. In: *Journal of Heuristics* 8.6 (2002), pp. 629–642. DOI: 10.1023/a:1020377910258. URL: https://doi.org/10.1023/a:1020377910258.

[19] J. F. Gonçalves and M. G. C. Resende. “A biased random key genetic algorithm for 2D and 3D bin packing problems”. In: *International Journal of Production Economics* 145.2 (2013), pp. 500–510. ISSN: 0925-5273. DOI: https://doi.org/10.1016/j.ijpe.2013.04.019. URL: https://www.sciencedirect.com/science/article/pii/S0925527313001837.

[20] J. F. Gonçalves and M. G. C. Resende. “A parallel multi-population biased random-key genetic algorithm for a container loading problem”. In: *Computers Operations Research* 39.2 (2012), pp. 179–190. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2011.03.009. URL: https://www.sciencedirect.com/science/article/pii/S0305054811000827.

[21] F. Gzara, S. Elhedhli, and B. Yildiz. “The Pallet Loading Problem: Three-dimensional Bin Packing with Practical Constraints”. In: *European Journal of Operational Research* 287 (June 2020). DOI: 10.1016/j.ejor.2020.04.053.

[22] F. Hutter, H. Hoos, and K. Leyton-Brown. “An Efficient Approach for Assessing Hyperparameter Importance”. In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. Bejing, China: PMLR, 22–24 Jun 2014, pp. 754–762. URL: https://proceedings.mlr.press/v32/hutter14.html.

[23] G. D. Israel. “Determining sample size”. In: (1992).

[24] J. Jozefowska et al. “Fast truck-packing of 3D boxes”. In: *Engineering Management in Production and Services* 10 (June 2018), pp. 29–40. DOI: 10.2478/emj-2018-0009.

[25] K. Klima. *Normality testing - skewness and Kurtosis*. Mar. 2021. URL: https://community.gooddata.com/metrics-and-maql-kb-articles-43/normality-testing-skewness-and-kurtosis-241.

[26] S. Kokoska and D. Zwillinger. *CRC standard probability and statistics tables and formulae*. Crc Press, 2000, p. 16.

[27] K. K. Lai and J. Chan. “Developing a simulated annealing algorithm for the cutting stock problem”. In: *Computers Industrial Engineering* 32 (Jan. 1997), pp. 115–127. DOI: 10.1016/S0360-8352(96)00205-7.

[28] A. H. Land and A. G. Doig. “An Automatic Method of Solving Discrete Programming Problems”. In: *Econometrica* 28.3 (1960), pp. 497–520. ISSN: 00129682, 14680262. URL: http://www.jstor.org/stable/1910129 (visited on 04/14/2022).

[29] A. Lim, B. Rodrigues, and Y. Wang. “A multi-faced buildup algorithm for three-dimensional packing problems”. In: *Omega* 31.6 (2003), pp. 471–481. ISSN: 0305-0483. DOI: https://doi.org/10.1016/j.omega.2003.08.004. URL: https://www.sciencedirect.com/science/article/pii/S030504830300094X.

[30] B. Mahvash, A. Awasthi, and S. Chauhan. “A column generation-based heuristic for the three-dimensional bin packing problem with rotation”. In: *Journal of the Operational Research Society* 69 (Mar. 2017). DOI: 10.1057/s41274-017-0186-7.

[31] S. Martello, D. Pisinger, and D. Vigo. “The Three-Dimensional Bin Packing Problem”. In: *Operations Research* 48 (Feb. 1998). DOI: 10.1287/opre.48.2.256.12386.

[32] M. Martin et al. “Three-dimensional guillotine cutting problems with constrained patterns: MILP formulations and a bottom-up algorithm”. In: *Expert Systems with Applications* 168 (2021), p. 114257. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2020.114257. URL: https://www.sciencedirect.com/science/article/pii/S0957417420309726.

[33] Y. Ozaki et al. “Multiobjective Tree-Structured Parzen Estimator for Computationally Expensive Optimization Problems”. In: GECCO ’20. Cancún, Mexico: Association for Computing Machinery, 2020, pp. 533-541. ISBN: 9781450371285. DOI: 10.1145/3377930.3389817. URL: https://doi.org/10.1145/3377930.3389817.

[34] C. Paquay, M. Schyns, and S. Limbourg. “Three dimensional Bin Packing Problem applied to air cargo”. In: 2011.

[35] J. W. Pratt. “Remarks on Zeros and Ties in the Wilcoxon Signed Rank Procedures”. In: *Journal of the American Statistical Association* 54.287 (1959), pp. 655-667. DOI: 10.1080/01621459.1959.10501526. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1959.10501526. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1959.10501526.

[36] C. Prud’homme, J.-G. Fages, and X. Lorca. “Choco solver documentation”. In: *TASC, INRIA Rennes, LINA CNRS UMR* 6241 (2016).

[37] K. Sörensen and F. Glover. “Metaheuristics”. In: Jan. 2013, pp. 960–970. ISBN: 978-1-4419-1137-7. DOI: 10.1007/978-1-4419-1153-7_1167.

[38] R. Verma et al. “A Generalized Reinforcement Learning Algorithm for Online 3D Bin-Packing”. In: (2020). arXiv: 2007.00463 [cs.AI].

[39] F. Wang and K. Hauser. “Stable Bin Packing of Non-convex 3D Objects with a Robot Manipulator”. In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 8698–8704. DOI: 10.1109/ICRA.2019.8794049.

[40] F. Wilcoxon. “Some Rapid Approximate Statistical Procedures”. In: *Annals of the New York Academy of Sciences* 52 (Dec. 2006), pp. 808–814. DOI: 10.1111/j.1749-6632.1950.tb53974.x.

[41] A. Ye. *The beauty of Bayesian optimization, explained in simple terms*. Oct. 2020. URL: https://towardsdatascience.com/the-beauty-of-bayesian-optimization-explained-in-simple-terms-81f3ee13b10f.

[42] A. Zudio et al. “BRKGA/VND Hybrid Algorithm for the Classic Three-dimensional Bin Packing Problem”. In: *Electronic Notes in Discrete Mathematics* 66 (2018). 5th International Conference on Variable Neighborhood Search, pp. 175-182. ISSN: 1571-0653. DOI: https://doi.org/10.1016/j.endm.2018.03.023. URL: https://www.sciencedirect.com/science/article/pii/S1571065318300696.

A
Side project: contaminated articles

The side project about contaminated articles can be found in the confidential version of the thesis.

B

Scholarly paper

B.1. Abstract

Three-dimensional Bin Packing, also known as the Multi-Container Loading Problem, has numerous real world applications and is thus very well researched in the scientific community. However, the data set used by state-of-the-art algorithms only accounts for items that have dimensions that are smaller than the dimensions of the bins in which they are placed. While current state-of-the-art algorithms can deal efficiently with these combinatorial challenges, they appear to not be sufficiently capable to handle real life constraints. In reality, there items that have one or more dimensions that are bigger than the bin. These items could still fit if they were to be diagonally rotated or squeezed. This paper proposes an extension of the state-of-the-art Three-dimensional Bin Packing algorithm, which is called a Biased Random-Key Genetic Algorithm. This extension serves to include single or double diagonal rotation and items squeezing, as to allow for these so-called oversized items to still be placed inside a bin. Furthermore, the inclusion of memoization, a different stopping criterion and priority based multithreading improves the computation time of the current state-of-the-art.

B.2. Outline

B.2.1. Introduction

A **Problem statement:** Current state-of-the-art performs well in generated data sets, but often do not take real life constraints into account. This paper focuses on placing items that have one or more dimensions bigger than a bin. It also introduces more techniques that lead to a reduction in computation time compared to the current state-of-the-art

B.2.2. Background

A **Current state-of-the-art:** BRKGA is currently the best performing Three-dimensional Bin Packing algorithm.

B **Research gaps:** Current state-of-the-art can not deal with items that have one or more dimensions bigger than a bin.

B.2.3. Algorithm

A **Bag stretching:** Bin stretching can lead to a reduction if multiple bins are combined into crates.

B **Fragility:** Fragility can be modeled by giving every item a fragility score and incorperating this score in the BRKGA when determining placement order.

C **Diagonal rotation:** Single and double diagonal rotation allows items that have one or more dimensions bigger than a bin to still fit.

D **Squeezing:** By squeezing items, they can be made to fit bins even if they have one or more bigger dimensions. For this to work it must be known whether items are squeezable.

E **Memoization:** Memoization leads to a reduction of computation time for BRKGA. It is also an essential part to let double diagonal rotation work in reasonable time.

F **(priority based) Multithreading:** While BRKGA already uses multithreading, the use of priority based multithreading allows for more high level parallelization which leads to less overhead.

B.2.4. Results

A **Feature expansion:** Show that bag stretching leads to a reduction in crates when bins are packed in crates (e.g., Picnic's bags to totes approach). Also show that BRKGA can not deal with oversized articles but the algorithm of this paper can.

B **Performance improvements:** Show that the performance improvements indeed lead to a reduction in computation time through ablation studies.

B.2.5. Conclusion & future work

Oversized articles: Current research has been using data sets that are used for decades which does not fully represent real life constraints. This paper aims to break this trend by accounting for oversized articles.

Improvements over existing research: This paper proposes memoization and (priority based) multithreading to further reduce computation time of the state-of-the-art.

Future work: For future work one could further expand upon BRKGA by including irregular items and stability. While both these features already exist in other works (as explained in Chapter 6), it hasn't been applied to the current state-of-the-art. At the end it should be repeated that this research field should move to more complex data sets, which better reflect real life constraints.

C
Oversized articles

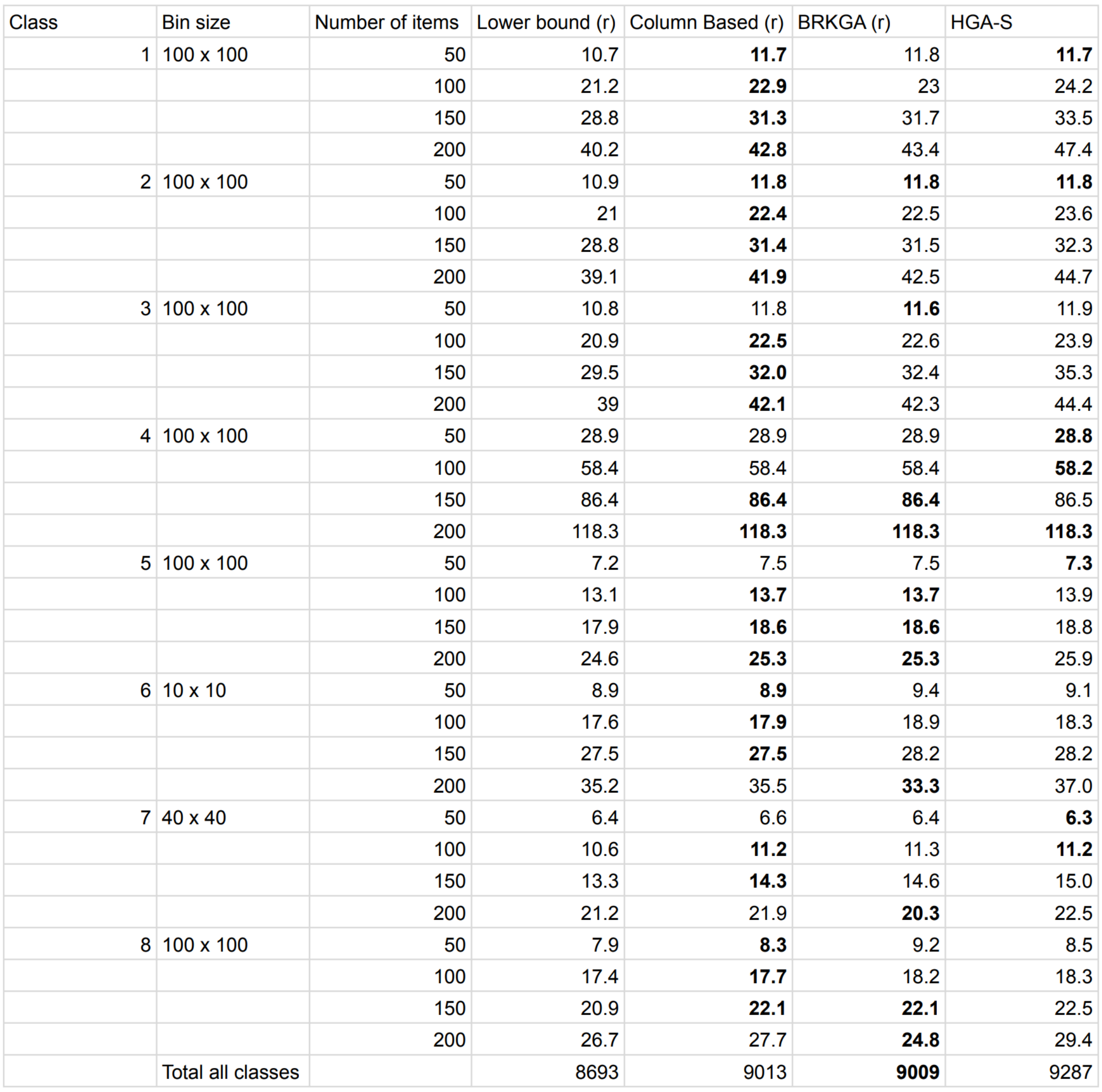
A list of oversized articles and their squeezability is provided in the confidential version of the thesis.

D
Squeezability of most frequently ordered articles

A list of the top 100 most sold articles per temperature zone and their squeezability is provided in the confidential version of the thesis.

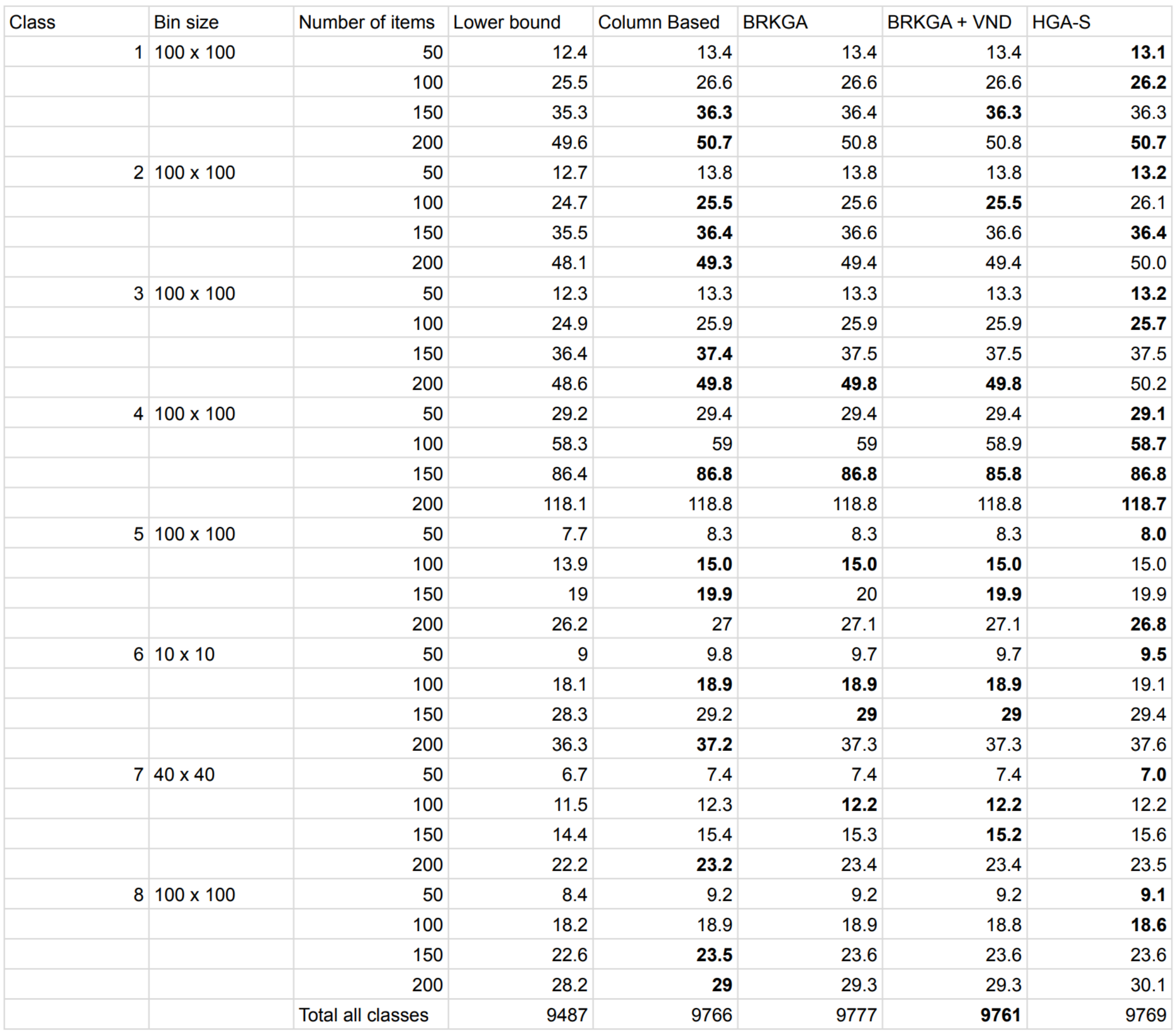
E

Detailed comparison of related work



Class	Bin size	Number of items	Lower bound (r)	Column Based (r)	BRKGA (r)	HGA-S
1	100 x 100	50	10.7	**11.7**	11.8	**11.7**
		100	21.2	**22.9**	23	24.2
		150	28.8	**31.3**	31.7	33.5
		200	40.2	**42.8**	43.4	47.4
2	100 x 100	50	10.9	**11.8**	**11.8**	**11.8**
		100	21	**22.4**	22.5	23.6
		150	28.8	**31.4**	31.5	32.3
		200	39.1	**41.9**	42.5	44.7
3	100 x 100	50	10.8	11.8	**11.6**	11.9
		100	20.9	**22.5**	22.6	23.9
		150	29.5	**32.0**	32.4	35.3
		200	39	**42.1**	42.3	44.4
4	100 x 100	50	28.9	28.9	28.9	**28.8**
		100	58.4	58.4	58.4	**58.2**
		150	86.4	**86.4**	**86.4**	86.5
		200	118.3	**118.3**	**118.3**	**118.3**
5	100 x 100	50	7.2	7.5	7.5	**7.3**
		100	13.1	**13.7**	**13.7**	13.9
		150	17.9	**18.6**	**18.6**	18.8
		200	24.6	**25.3**	**25.3**	25.9
6	10 x 10	50	8.9	**8.9**	9.4	9.1
		100	17.6	**17.9**	18.9	18.3
		150	27.5	**27.5**	28.2	28.2
		200	35.2	35.5	**33.3**	37.0
7	40 x 40	50	6.4	6.6	6.4	**6.3**
		100	10.6	**11.2**	11.3	**11.2**
		150	13.3	**14.3**	14.6	15.0
		200	21.2	21.9	**20.3**	22.5
8	100 x 100	50	7.9	**8.3**	9.2	8.5
		100	17.4	**17.7**	18.2	18.3
		150	20.9	**22.1**	**22.1**	22.5
		200	26.7	27.7	**24.8**	29.4
	Total all classes		8693	9013	**9009**	9287

Figure E.1: Total amount of bins with rotation.



Class	Bin size	Number of items	Lower bound	Column Based	BRKGA	BRKGA + VND	HGA-S
1	100 x 100	50	12.4	13.4	13.4	13.4	**13.1**
		100	25.5	26.6	26.6	26.6	**26.2**
		150	35.3	**36.3**	36.4	**36.3**	36.3
		200	49.6	**50.7**	50.8	50.8	**50.7**
2	100 x 100	50	12.7	13.8	13.8	13.8	**13.2**
		100	24.7	**25.5**	25.6	**25.5**	26.1
		150	35.5	**36.4**	36.6	36.6	**36.4**
		200	48.1	**49.3**	49.4	49.4	50.0
3	100 x 100	50	12.3	13.3	13.3	13.3	**13.2**
		100	24.9	25.9	25.9	25.9	**25.7**
		150	36.4	**37.4**	37.5	37.5	37.5
		200	48.6	**49.8**	**49.8**	**49.8**	50.2
4	100 x 100	50	29.2	29.4	29.4	29.4	**29.1**
		100	58.3	59	59	58.9	**58.7**
		150	86.4	**86.8**	**86.8**	**85.8**	**86.8**
		200	118.1	118.8	118.8	118.8	**118.7**
5	100 x 100	50	7.7	8.3	8.3	8.3	**8.0**
		100	13.9	**15.0**	**15.0**	**15.0**	15.0
		150	19	**19.9**	20	**19.9**	19.9
		200	26.2	27	27.1	27.1	**26.8**
6	10 x 10	50	9	9.8	9.7	9.7	**9.5**
		100	18.1	**18.9**	**18.9**	**18.9**	19.1
		150	28.3	29.2	**29**	**29**	29.4
		200	36.3	**37.2**	37.3	37.3	37.6
7	40 x 40	50	6.7	7.4	7.4	7.4	**7.0**
		100	11.5	12.3	**12.2**	**12.2**	12.2
		150	14.4	15.4	15.3	**15.2**	15.6
		200	22.2	**23.2**	23.4	23.4	23.5
8	100 x 100	50	8.4	9.2	9.2	9.2	**9.1**
		100	18.2	18.9	18.9	18.8	**18.6**
		150	22.6	**23.5**	23.6	23.6	23.6
		200	28.2	**29**	29.3	29.3	30.1
		Total all classes	9487	9766	9777	**9761**	9769

Figure E.2: Total amount of bins without rotation.