SEMINAR - ALBERT HEIJN CASE

ERASMUS UNIVERSITEIT ROTTERDAM

GROUP 3

MARCH 15, 2022

Solving a weighted 3D-Bin Packing Problem with fragile objects using a 'Biased Random-Key Genetic Algorithm'

Authors	*Student numbers*
BOOSMAN, Wessel	589273
[redacted] Eric	[redacted]
[redacted] Xander	[redacted]
[redacted] Yasemin	[redacted]

Abstract

This paper presents a Biased Random-Key Genetic Algorithm as its primary solution approach to the 3D-BIN PACKING PROBLEM with weight constraints. The problem is to optimise the packing process of the grocery delivery branch of a major Dutch supermarket chain, Albert Heijn. We conclude that the Genetic Algorithm (GA) provides good quality solutions within a reasonable amount of time, especially when compared to an exact solution approach and comparative heuristics. The GA provided an optimality gap of 7% on a data-set which reflects actual grocery shopping behaviour of 1000 consumers. Methods specifically designed to improve the running time of the GA are introduced and a clear trade-off between computational time and solution quality is found. In order to meet the practical characteristics of grocery delivery services, two fragility extensions are introduced to prevent fragile groceries from being damaged. The extension causes a 5% increase in optimality gap compared to the original model, however it does prove to be viable for this specific problem and can be well incorporated in the GA.

Contents

1 Introduction

The immense growth of e-commerce has lead to a society in which groceries are ordered online, a development that is still going on. As a result of this demand, major Dutch supermarket chains are competing in grocery delivery services. The viability of these services depend on the reduced willingness of people to go outside as well as the optimisation of all stages within the the delivery service, such as efficient route building for delivery trucks, or minimising packing time of an order in the warehouse. This paper concerns the issue of packing groceries, referred to as items, in delivery crates, such that the total number of crates necessary to fulfil an order is minimised. Optimising this stage of the delivery service should have a positive effect on the total number of delivery trucks and thus the total cost. This problem corresponds to the well researched 3D-Bin Packing Problem (3D-BPP) with an additional weight limit constraint, due to the structural limitations of the delivery crates.

The one-dimensional bin-packing problem is a known $\mathcal{NP}$-hard problem. Since the three-dimensional case is a generalisation of the one-dimensional case, the 3D-BPP is $\mathcal{NP}$-hard as well. As a result, it is safe to assume that describing and solving the 3D-BPP as a combinatorial optimisation problem will become practically impossible for sufficiently large orders, even with state-of-the-art commercial solvers.

This fact gives motivation for alternative solution approaches for which the run time does not grow exponentially in the order size. In Gonçalves and Resende (2013) a Biased Random Key Genetic Algorithm (BRKGA) is proposed as a solution approach to the 3D-BPP. The advantage of this algorithm is that you eliminate an enormous number of decision variables from the equation, such that solutions are found within a reasonable amount of time. Due to the promising results the authors present in their paper, this method is used as a solution approach in the current paper. To give an indication on the performance of the BRKGA, we introduce two heuristics to compare it with. It turns out that the BRKGA gives high quality solutions in a reasonable amount of time, and is able to produce an optimality gap of about 7%. In Chapter 5, adjustments to the algorithm are introduced to further reduce the computation time.

As an extension to the Genetic Algorithm (GA) mentioned above, a model is introduced which prevents situations where heavy (or 'non-fragile') items are placed on top of 'fragile' items. In Section 4.3 we introduce the assumptions we make to incorporate the fragility of the items into the algorithm. It follows that the extended BRKGA still solves the problem within a reasonable amount of time, although it is considerably slower compared to not taking fragility into account. Furthermore, including the fragility constraints negatively affects the solution quality in terms of the number of crates. However, for our specific optimisation problem it is still valuable to take the fragility into consideration.

In the remainder of this paper, we will start with a literature review in Chapter 2. The problem description and an analysis of the data used in this paper is given in Chapter 3. In Chapter 4 we will explain all methods that are used for this case study. This chapter includes the exact formulation of the problem, the genetic algorithm, the fragility and heuristics used for comparison. Adjustments that were made in order to decrease the computation times of the algorithms are presented in Chapter 5. Chapter 6 is dedicated to the numerical results of all heuristics and algorithms. Finally, in Chapters 7 and 8, the conclusion and discussion are given respectively.

2 Theoretical Framework

Multiple practical challenges describe the same kind of problem we are facing within the framework of this case study, which aims to maximise the utility of crates for a major grocery delivery service. Within the literature different terms are used for crates, such as bins or containers, based on the specifics of the problem. In an article by Wäscher et al. (2007) the authors introduce a formal definition on the structure of packing problems. They state that a packing problem has to consist of at least the following properties. First, the problem must contain large objects (in our case crates) as well as smaller items (in our case the grocery items to be packed). Furthermore, the 'geometric condition' has to be satisfied, which states that a subset of small items has to be inside a large object such that (*i*) the small items are entirely inside the large object, and (*ii*) the small items do not overlap. Since the problem we are considering within this case study satisfies this definition, it can be seen as a packing problem. Still, various problems discussed in literature satisfy the definition mentioned above yet they could significantly differ in terms of constraints or the objective functions.

For example, the Vehicle Loading Problem (VLP) as described by Liu et al. (2016) considers items of different shapes, sizes, orientations and profits which must be assigned to a vehicle. The objective in the VLP is to maximise the profit, which is different from the objective in our case study.

Another packing problem is described by Pisinger (2002) where the authors consider the Container Loading Problem (CLP). The focus within this problem is to maximise the fill-rate within one container, which is very similar to the objective in our case study. However, the CLP focuses on optimising one container, while our objective is to minimise the number of containers (crates). A variation of this problem is allowing for more than one container as is done in the Multi Container Loading Problem (MCLP) described by Chen et al. (1995). However, in this problem containers are allowed to differ from each other in size and the objective is to find a subset of containers such that the shipping costs are minimised. Again, this problem touches upon the problem we consider, yet small variations make a one-to-one comparison impossible.

A generalisation of the MCLP is the 3D-Bin Packing Problem, where the containers have a fixed size and the objective is to minimise the number of containers. So the problem of our case study can be considered a 3D-BPP with an additional constraint imposing a weight limit on the crates. The 3D-BPP is a widely documented problem with solution approaches ranging from machine learning algorithms to integer linear programming and every logically constructed heuristic in between. To the best of our knowledge, Martello et al. (2000) was the first to introduce an exact approach to the 3D-BPP. Future work build upon this paper is, among others, done by Hifi et al. (2010) where the authors propose methods for improving the lower bound of a solution. Moreover, in a paper by Hu et al. (2017) the authors introduce the idea of combining an exact formulation with reinforcement learning. Still, both papers exclude the weight limit of a crate. Besides that, in Hu et al. (2017) the authors consider a different objective function which minimises the surface area of the crates and in Hifi et al. (2010) different orientations of the items are ignored.

Since the 3D-BPP is $\mathcal{NP}$-hard, approximation algorithms and heuristics are presented in various papers to solve the problem. For example, within the work of Crainic et al. (2008) the authors introduce the Extreme Point Based Heuristic. This is a constructive heuristic which focuses on the placement of an item, using so-called extreme points, extending the corner points concept introduced by Martello et al. (2000).

Another heuristic is the Peak Filling Slice Push Heuristic described by Maarouf et al. (2008). They split the crates into slices with same length and height and fill every slice separately. After that, the filled slices are compressed such that the empty space inside a crate is minimised. This method could improve the running time, since the volume of each sub-space is much smaller than the complete space, therefore limiting the number of possible combinations.

The Guided Local Search (GLS) Heuristic presented by Faroe et al. (2003) starts with a greedy heuristic to find an upper bound to the problem. Afterwards, a local search heuristic is used to search for a tighter upper bound by iteratively decreasing the number of crates and trying to reallocate the items.

One of the main solution approaches used in this paper is based on the paper of Gonçalves and Resende (2013). The authors present a genetic algorithm, which is based on random keys, to solve 2D and 3D bin-packing problems. Within this method they introduce two heuristics for the placement of items, such that the items are packed as dense as possible. Furthermore, the algorithm is compared to other methods and heuristics published in literature, including the GLS Heuristic presented by Faroe et al. (2003), the Tabu Search Heuristic by Lodi et al. (2002) and a GRASP Heuristic by Parreño et al. (2010). The authors claim that their genetic algorithm outperforms all the approaches it is compared to.

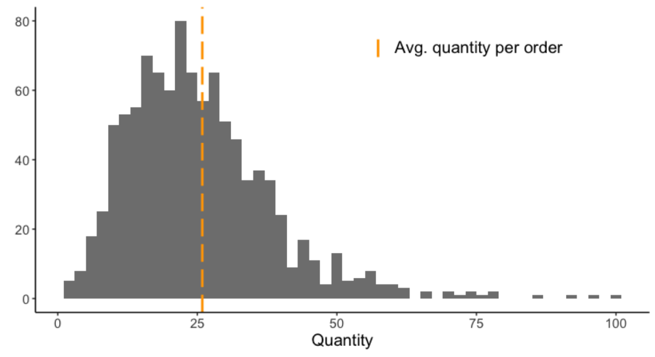
It is obvious that there are various extensions possible to the 3D-BPP as each problem has its own restrictions and characterisations. Since our case study focuses on packing grocery items, the fragility of the items could be taken into consideration. In Clautiaux et al. (2014) an exact formulation for a bin-packing problem with fragile objects is given. Using a column generation algorithm they present lower bounds to the problem and with the help of a constructive heuristic in combination with a Variable Neighbourhood Search algorithm upper bounds are presented. However this paper only focuses on one-dimensional bin-packing problems. To the extent of our knowledge, taking fragility into account in three-dimensional multi-item packing problems, specifically together with a genetic algorithm, has not been researched yet. It may still be beneficial in multiple optimisation problems, with grocery packing for delivery services being just one of them. Moreover, we guess that container and truck loading companies face the same kind of problems.

3 Problem Description

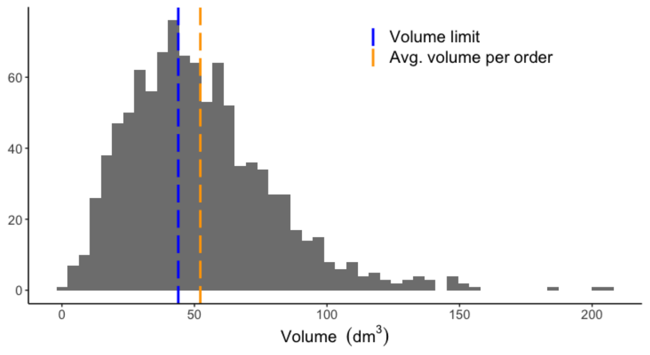
This section is dedicated to a detailed description of the problem and an analysis of the provided data. The 3D-Bin Packing Problem considered in this case study consists of multiple orders. Each order k consists of n_k cuboid shaped items i (with $i = 1, \ldots, n_k$ and $k = 1, \ldots, K$ with K the number of orders) with corresponding weight g_i, length l_i, width w_i and height h_i. Each order has to be packed in crates with fixed length L, width W and height H. Our goal is to minimise the number of crates necessary to pack an order, while ensuring that all items assigned to a crate, are completely inside that crate, and the cumulative weight of those items do not exceed the weight limit G. Finally, items are not allowed to overlap.

3.1 Data Analysis

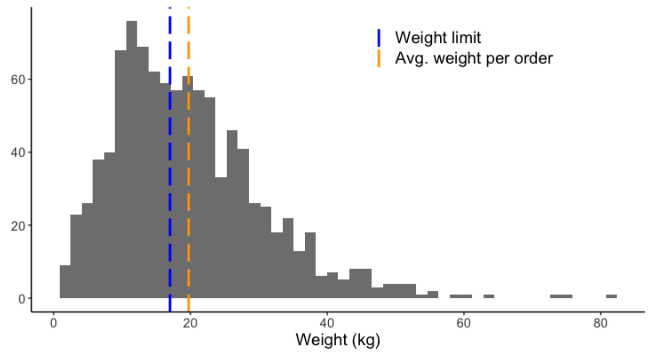
The data for this case study is provided by Albert Heijn (AH) and contains a set of orders and a set of items. The items represent actual products available in their supermarket, whilst the orders are generated randomly by AH (due to the number of available items), but still represent orders made in reality. In total there are 1000 orders and 500 items. With the help of the programming language `R-Studio` (R Core Team (2019)) an analysis is made of the data. In Figure 1 three histograms are presented which show the quantity, volume and weight per order.



(a) Histogram of item quantity per order



(b) Histogram of total volume per order



(c) Histogram of total weight per order

Figure 1: An analysis of the orders, considering the number of items, total volume and total weight.

The average volume, weight and number of items per order are summarised in Table 1.

Table 1: Average volume, weight and quantity per order

Avg. volume (L)	Avg. weight (kg)	Avg. quantity
52.22	25.94	26

Note that an order can contain a specific item multiple times and therefore, items in an order do not need to be unique. The specifics of a crate are summarised in Table 2.

Table 2: Dimensions, volume (V) and weight limit of a crate

L (mm)	W (mm)	H (mm)	V (L)	G (kg)
501	321	273	43.9	17

Another interesting property is the minimum required number of crates to pack an order, i.e. what is a possible lower bound to the problem? We have two main restrictions, namely the total weight and volume per crate. The minimum number of needed crates per order k by only considering the weight or volume is $\frac{G_k}{G}$ or $\frac{V_k}{V}$, respectively, with G_k defined as the total weight of all items in order k and V_k the total volume of all items in order k. Taking the ceiling function over the maximum of these two, so:

$$\left\lceil \max\left\{\frac{V_k}{V}, \frac{G_k}{G}\right\}\right\rceil,$$

gives us an obvious lower bound to the problem. Table 3 summarises for how many orders the lower bound is determined by the maximum volume restriction and for how many orders the weight limitation is decisive. Note that the number of crates has to be integer. That is why we also consider the case of the ceiling function. We see that in both cases the volume is more restrictive for determining the lower bound.

Table 3: Number of orders which have the volume or weight as a more restrictive factor

	$\frac{V_k}{V} > \frac{G_k}{G}$	$\frac{V_k}{V} < \frac{G_k}{G}$	$\left\lceil\frac{V_k}{V}\right\rceil > \left\lceil\frac{G_k}{G}\right\rceil$	$\left\lceil\frac{V_k}{V}\right\rceil < \left\lceil\frac{G_k}{G}\right\rceil$
Number of orders	553	447	87	64

4 Methodology

This section is completely dedicated to the methods used to solve the 3D-BPP. First, an exact formulation of the 3D-BPP with weight limits will be given. Next, we propose a genetic algorithm to tackle the problem. As an extension, we introduce fragility as a practical restriction on the problem and we discuss how we could change the genetic algorithm such that it incorporates this limitation. Lastly, we present two 'fast' heuristics, which are constructed based on common sense and serve as proper comparisons to the more advanced genetic algorithm.

4.1 Exact Formulation

The 3D-Bin Packing Problem can be formulated as an integer linear programming (ILP) problem. In this section we will define the variables, constraints and objective function used for the formulation of the ILP problem. Note that the exact formulation of the problem considers only one order.

4.1.1 Variables

Let β be the number of crates which will be minimised and let n be the number of items of the order. The label β_i denotes the label of the crate in which item i is placed. These labels will be in increasing order starting from 1, so $\beta_i \geq 1$. Thus we can define β as the largest label of the used crates, so:

$$\beta := \max_{1 \leq i \leq n} \beta_i.$$

Let (x_i, y_i, z_i) be the coordinates of the left bottom front (LBF) corner of item i, with $x_i, y_i, z_i \geq 0$. The LBF coordinate of a crate is defined as $(0, 0, 0)$. Then we define (x_i, y_i, z_i, β_i) as the location of item i in the crate with label β_i. In this case study, all six possible orientations for a rectangular prism shaped item i are considered. The combination of an item with its orientation is denoted by φ_{ik}, with $k = 1, \ldots, 6$. The different orientations are defined as follows:

Table 4: Orientations of item i, where its length, width and height are along the x-, y- or z-axis.

	φ_{i1}	φ_{i2}	φ_{i3}	φ_{i4}	φ_{i5}	φ_{i6}
Length	x	z	x	z	y	y
Width	y	y	z	x	x	z
Height	z	x	y	y	z	x

A list of all used variables is given below.

The *input variables* of the problem are given by:

- L: the length of a crate
- W: the width of a crate
- H: the height of a crate
- G: the maximum allowed weight of a crate
- l_i: the length of item i
- w_i: the width of item i
- h_i: the height of item i
- g_i: the weight of item i
- $\mathcal{B}$: an upper bound for the number of crates

There are also a few *additional variables* which are given by:

- $\hat{l}_i$: length of item i after orientation
- $\hat{w}_i$: width of item i after orientation
- $\hat{h}_i$: height of item i after orientation

The *decision variables* of the problem are given by:

- β: largest label of the used crates
- x_i: the LBF coordinate of item i in x-axis
- y_i: the LBF coordinate of item i in y-axis
- z_i: the LBF coordinate of item i in z-axis
- c_{ij}: indicates whether item i is in a crate with a smaller label than item j, so $\beta_i < \beta_j$
- k_{ij}: indicates whether item i is on the left of item j or not
- u_{ij}: indicates whether item i is under item j or not
- b_{ij}: indicates whether item i is in the back of item j or not
- φ_{ik}: indicates whether orientation k is used for item i, with $k = 1 \ldots, 6$,

where c_{ij}, k_{ij}, u_{ij}, b_{ij} and φ_{ik} are binary variables. The additional variables $\hat{l}_i$, $\hat{w}_i$ and $\hat{h}_i$ are the variable length, width and height of an item and depend on the orientation.

4.1.2 Integer Linear Programming Problem

The formulation of the ILP problem can be given as follows:

$$
\begin{aligned}
\min \quad & \beta && && (1)\\
\text{s.t.} \quad & k_{ij} + k_{ji} + u_{ij} + u_{ji} + b_{ij} + b_{ji} + c_{ij} + c_{ji} = 1 && i, j = 1, \ldots, n \wedge i < j && (2)\\
& x_i - x_j + L(k_{ij} - c_{ij} - c_{ji}) \leq L - \hat{l}_i && i, j = 1, \ldots, n \wedge i \neq j && (3)\\
& y_i - y_j + W(b_{ij} - c_{ij} - c_{ji}) \leq W - \hat{w}_i && i, j = 1, \ldots, n \wedge i \neq j && (4)\\
& z_i - z_j + H(u_{ij} - c_{ij} - c_{ji}) \leq H - \hat{h}_i && i, j = 1, \ldots, n \wedge i \neq j && (5)\\
& (\mathcal{B} - 1)(k_{ij} + k_{ji} + u_{ij} + u_{ji} + b_{ij} + b_{ji}) + \beta_i - \beta_j + \mathcal{B}c_{ij} \leq \mathcal{B} - 1 && i, j = 1, \ldots, n \wedge i \neq j && (6)\\
& \hat{l}_i = \varphi_{i1} l_i + \varphi_{i2} h_i + \varphi_{i3} l_i + \varphi_{i4} w_i + \varphi_{i5} w_i + \varphi_{i6} h_i && i = 1, \ldots, n && (7)\\
& \hat{w}_i = \varphi_{i1} w_i + \varphi_{i2} w_i + \varphi_{i3} h_i + \varphi_{i4} h_i + \varphi_{i5} l_i + \varphi_{i6} l_i && i = 1, \ldots, n && (8)\\
& \hat{h}_i = \varphi_{i1} h_i + \varphi_{i2} l_i + \varphi_{i3} w_i + \varphi_{i4} l_i + \varphi_{i5} h_i + \varphi_{i6} w_i && i = 1, \ldots, n && (9)\\
& \varphi_{i1} + \varphi_{i2} + \varphi_{i3} + \varphi_{i4} + \varphi_{i5} + \varphi_{i6} = 1 && i = 1, \ldots, n && (10)\\
& \sum_{j \neq i} g_j (k_{ij} + k_{ji} + u_{ij} + u_{ji} + b_{ij} + b_{ji}) \leq G - g_i && i = 1, \ldots, n && (11)\\
& 0 \leq x_i \leq L - \hat{l}_i && i = 1, \ldots, n && (12)\\
& 0 \leq y_i \leq W - \hat{w}_i && i = 1, \ldots, n && (13)\\
& 0 \leq z_i \leq H - \hat{h}_i && i = 1, \ldots, n && (14)\\
& 1 \leq \beta_i \leq \beta \leq \mathcal{B} && i = 1, \ldots, n && (15)\\
& k_{ij}, u_{ij}, b_{ij}, c_{ij} \in \{0, 1\} && i, j = 1, \ldots, n && (16)\\
& \varphi_{i1}, \varphi_{i2}, \varphi_{i3}, \varphi_{i4}, \varphi_{i5}, \varphi_{i6}, \in \{0, 1\} && i = 1, \ldots, n && (17)\\
& \beta_i, \beta \in \mathbb{N}^+ && i = 1, \ldots, n && (18)\\
& x_i, y_i, z_i \in \mathbb{N} && i = 1, \ldots, n && (19)\\
& \hat{l}_i, \hat{w}_i, \hat{h}_i \in \mathbb{N} && i = 1, \ldots, n. && (20)
\end{aligned}
$$

The ILP formulation above is largely inspired on the articles by Hifi et al. (2010) and Hu et al. (2017). In fact, the objective function and most of the constraints are adopted from one of these papers. Only constraints (11) are our own contribution to this formulation.

4.1.3 Constraints

Constraints (2), (3), (4) and (5) ensure that no overlapping occurs between two items in a crate. When one of the variables k_{ij}, k_{ji}, u_{ij}, u_{ji}, b_{ij} or b_{ji} is equal to 1, it follows that (according to constraints (2)) the variables c_{ij} and c_{ji} have to be equal to zero. Subsequently, from the definition of c_{ij} we have $\beta_i \geq \beta_j$ and $\beta_j \geq \beta_i$ and thus $\beta_i = \beta_j$. This implies that item i and j are in the same crate. Constraints (6) link all the variables from constraints (2) to β_i and β_j variables. By making smart use of a valid upper bound $\mathcal{B}$, it forces the crates to have different labels when items i and j are in different crates. Constraints (7), (8) and (9) give the length, width and height of item i after orientating, and constraints (10) ensures only one orientation per item is considered. The 3D-BPP including weight constraints are rarely described in the literature. Therefore, we introduce constraints (11), which guarantee that the maximum weight of a crate is not exceeded. Constraints (12), (13) and (14) are necessary to make sure that all of the items are completely inside a crate. Finally, constraints (16)-(20) represent the binary and integrality constraints.

4.2 Genetic Algorithm

This section includes a detailed description of the implemented Genetic Algorithm (GA). The first subsection is dedicated to the general principles of the GA. Subsequently, multiple placement heuristics are described. After that, we introduce the considered parameters, which affect the performance of the algorithm, and stopping criteria, indicating when the algorithm will terminate.

4.2.1 Biased Random-Key Genetic Algorithm

The current paper will use a GA to solve the 3D-BPP. A GA allows for searches through extremely large solution spaces that are too big for any exhaustive methods (Mitchell, 1998). As stated by Mitchell (1998), there is no explicit structure for a GA. Still, in general it consists of the following elements: a population of chromosomes, selection using a fitness function, a crossover method for offspring, and mutations (Mitchell (1998) & Snaselova and Zboril (2015)), which we will touch upon in the Sections 4.2.1.1, 4.2.1.2 and 4.2.1.3. The GA described in the current paper is almost identical to the one proposed in Gonçalves and Resende (2013). They presented a *Biased Random Key Genetic Algorithm* (BRKGA). In the remainder of this section we provide a clear overview on the most important parts of the BRKGA and the differences in methods compared to the original paper.

4.2.1.1 Population

The population of a GA is composed of individual solutions, where the initial population marks the start. Individual solutions, henceforth referred to as chromosomes, represent a solution to the 3D-BPP for a certain order. Intuitively, the solution would simply be the number of crates assigned to the selected order as this is the objective to minimise. However, using this representation would make it impossible for the GA to improve the current solution quality in forthcoming generations. Consequently, chromosomes are represented by two vectors consisting of random numbers ('Random Keys') on the domain $[0, 1]$. Each vector consists of n components, where n is the number of items in an order. The first vector corresponds to the sequence of the placement of the items ('Sequence Key'), while the second vector contains information on the orientation of the item ('Orientation Key'). Sorting the sequence key, directly represents the order in which the items will be placed. On the other hand, the orientation vector does not directly imply the orientation in which the items will be placed. Instead, it is used in the placement heuristics from the algorithm in (Gonçalves and Resende, 2013). Due to how the solution is 'encrypted' in a chromosome, feasibility of a chromosome is always ensured. However, this does require extra steps to determine the actual solution. As in Gonçalves and Resende (2013), we use a decoder to transmute the encrypted chromosome information into an actual solution. This is done through the use of placement heuristics, which are described later on in Section 4.2.2.

The initial population is randomly generated, thus all the keys in each chromosome are chosen at random. Note that each of these randomly created chromosomes corresponds to a feasible solution, since it is always allowed to open a new crate and all items have at least one feasible orientation in which they fit in a crate.

4.2.1.2 Selection, Mutation, Crossover

This part explains how the algorithmic parallelism of crossover, selection and mutation can be used to create new populations as proposed by Gonçalves and Resende (2013). Each population consists of an elite group of chromosomes, which represent the best solutions of the current population. The entire elite group is copied over to the new population in the next generation (*'selection'*). Hence, the best solution within a population is always equal to or better than the one of the previous population. Furthermore, to diversify the search space, randomness is introduced in the new population by adding mutated chromosomes (*'mutation'*). These *'mutants'* are new randomly constructed chromosomes. The new population is completed through offspring, generated by the current population. Offspring is created by using two chromosomes from the current population, the first parent from the elite group and the second from the non-elite group. Both are selected at random within each group, respectively. Subsequently, the value of each component in offspring's vectors is either equal to the component of the elite parent with probability ρ_e or the non-elite parent with probability $1-\rho_e$ (*'crossover'*). Two parent chromosomes always create one offspring chromosome and parent chromosomes can be selected more than once. As mentioned by Gonçalves and Resende (2013), there is no limit to the number of times a chromosome can be selected. In addition, the new populations are 'biased', since the selection is made by always using an elite parent.

4.2.1.3 Fitness

Each chromosome is attributed a fitness score, a measure to determine the quality of the solution. Gonçalves and Resende (2013) propose the use of a fitness score in which the number of crates is combined with the fill rate of the least filled crate in the solution. As the current problem is a minimisation problem, a lower fitness represents a better solution by either using less crates or packing the same crates more densely. The fitness function introduced by Gonçalves and Resende (2013) is stated below in Equation (21), where NB represents the number of crates.

$$\text{fitness} = \text{NB} + \frac{\text{Least filled crate in solution}}{\text{Total volume of a single crate}} \tag{21}$$

The current problem also incorporates a weight limit per crate. But according to the data analysis in Section 3.1, the weight limit is less likely to be the decisive factor when determining which crate to place an item in (see Table 3). Furthermore, in constraints (3)-(5) and (12)-(14) in Section 4.1, we see that for the limitations on volume a lot of constraints are required, while the weight limitation only considers constraints (11). Thus in the exact model, the volume is also more restrictive as we consider three dimensions compared to the single dimension of the weight. Hence, we preserve the use of the original fitness function.

4.2.2 Placement Heuristics

Subsection 4.2.1.1 provided an explanation on the architecture of a chromosome, which represents the solution indirectly. Therefore, the chromosome has to be decoded and transmuted into an actual solution. As mentioned before, the first part of a chromosome states the packing sequence of the items in a solution.

This is the first part of decoding the chromosome. The current section further explains the decoding procedure, in particular the use of the orientation vector and placement heuristics.

4.2.2.1 Empty Maximal Space

Before diving into the placement of items, we first introduce the definition of an 'Empty Maximal Space' (EMS), as proposed by Gonçalves and Resende (2013). An EMS is a maximum sized rectangular prism we can fit in an empty part of the crate such that is does not interfere with any of the placed items. The complete empty space within a crate can be described by a set of EMSs called $\mathbb{S}$.

Like all cuboid shaped spaces, an EMS can be represented by two coordinates. Namely, its origin, or the Back-Bottom-Left corner, and the corner furthest away form the origin, the Front-Top-Right corner. When an item is placed, at least one EMS in the crate is affected by this item. For that reason, we need to update the set $\mathbb{S}$ and possibly create new EMSs each time an item is placed in a crate. The method responsible for the creation of new EMSs is called the 'difference process' and is adopted from Lai and Chan (1997). The definition of the difference process is repeated in Definition 1 and is used for the creation of all new possible EMSs after an item is placed in a crate. An example of applying the difference process is depicted in Figures 2a through d.

Remark 1 *After adding item i with origin coordinates $(x_{O_i}, y_{O_i}, z_{O_i})$ and Front-Top-Right corner coordinates $(x_{F_i}, y_{F_i}, z_{F_i})$ to crate B, the set $\mathbb{S}$ of EMSs gets updated by applying the difference process between the item and each EMS in $\mathbb{S}$.*

Definition 1 *The* **difference process**, *as introduced by* Lai and Chan (1997), *is defined as the difference between an EMS and an item. Let item i be defined by the coordinates $(x_{O_i}, y_{O_i}, z_{O_i})$ and $(x_{F_i}, y_{F_i}, z_{F_i})$, and EMS E be defined by the coordinates $(x_{O_E}, y_{O_E}, z_{O_E})$ and $(x_{F_E}, y_{F_E}, z_{F_E})$. Then the possible set of new spaces created from the EMS E are given by:*

$$[(x_{O_E}, y_{O_E}, z_{O_E}), (x_{O_i}, y_{F_E}, z_{F_E})] \tag{22}$$
$$[(x_{F_i}, y_{O_E}, z_{O_E}), (x_{F_E}, y_{F_E}, z_{F_E})] \tag{23}$$
$$[(x_{O_E}, y_{O_E}, z_{O_E}), (x_{F_E}, y_{O_i}, z_{F_E})] \tag{24}$$
$$[(x_{O_E}, y_{F_i}, z_{O_E}), (x_{F_E}, y_{F_E}, z_{F_E})] \tag{25}$$
$$[(x_{O_E}, y_{O_E}, z_{O_E}), (x_{F_E}, y_{F_E}, z_{O_i})] \tag{26}$$
$$[(x_{O_E}, y_{O_E}, z_{F_i}), (x_{F_E}, y_{F_E}, z_{F_E})] \tag{27}$$

Using Remark 1 and Definition 1 we can generate the entire set of new EMSs after placing an item, derived from the old $\mathbb{S}$, as proposed by Lai and Chan (1997). After the creation of all new EMSs they also remove infinitely thin and fully enclosed EMSs, as stated in Rule 1 and 2. Nevertheless, this still results in updating the entire set $\mathbb{S}$. We actually only need to apply the difference process to the EMSs which interfere with the placed item. Applying the difference process to an EMS that does not interfere with the placed item will simply return itself again. Here, interfere means that an item is placed (partially) in an EMS. We can explain this by the fact that an EMS describes the **maximum** space within a crate where an item can be placed. An EMS can therefore never become larger after adding an item and an EMS cannot become smaller after placing an item if it does not interfere with the EMS. Therefore, we introduce Rule 3.

Rule 1 *If the origin of the created EMSs by the method described in* Definition 1 *and the Front-Top-Right corner of that EMS share at least one of the three coordinates, the EMS reduces to a two-dimensional or even one-dimensional space and is therefore infinitely thin. Infinitely thin EMS's are removed.*

Rule 2 *If a created EMS is as a whole inscribed by another EMS, the former one is removed.*

Rule 3 *After placing an item in a crate only EMSs that interfere with the item are updated. Earlier created EMSs that do not interfere with the item remain unchanged.*

Rule 3 does not alter the size of $\mathbb{S}$ after placing an item, it only strongly reduces the number of newly created EMSs. Moreover, as in Gonçalves and Resende (2013), we can further decrease the size of $\mathbb{S}$ by eliminating all EMSs from $\mathbb{S}$ whose volume or smallest dimension is smaller than the smallest volume or smallest dimension of the remaining items respectively. Rules 4 and 5 formally introduce these updates.

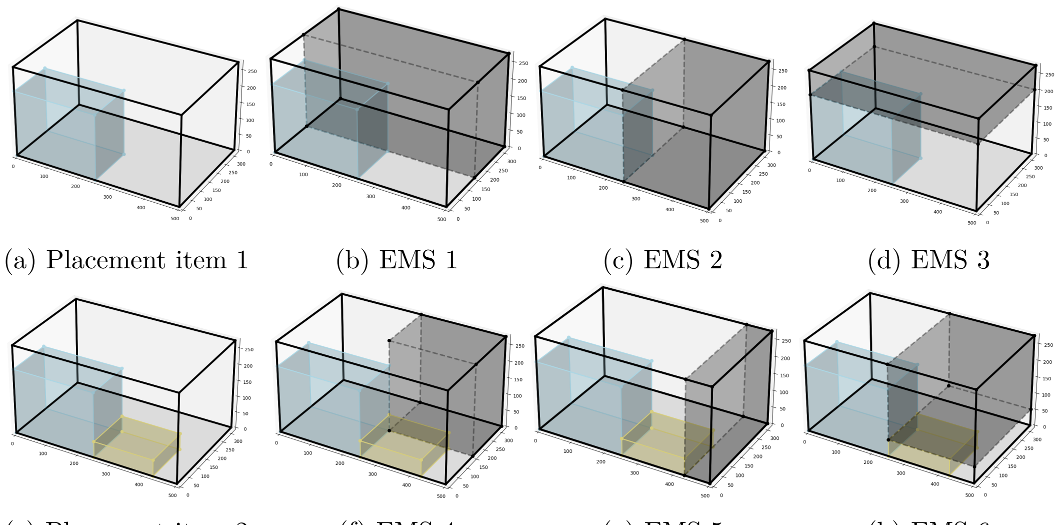
Rule 4 *If an EMS has a volume smaller than the smallest volume of the remaining items, the EMS is removed.*

Rule 5 *If an EMS's shortest dimension is smaller than the smallest dimension of each of the remaining items, the EMS is removed.*

Example

Below we will illustrate how EMSs are created and updated according to the previously defined rules. The order we will consider consists of two items. As the entire crate is equal to the first EMS, the first item will always be placed in the Back-Bottom-Left corner of the crate, as per definition of the placement heuristics. These heuristics will be explained in the two upcoming sections. It is visualised in Figure 2a. After placing item 1 there is only one EMS that needs to be updated, which is done according to the difference process as described in Definition 1. The difference process creates six new EMSs although only three of them are maintained, as can be seen in Figures 2b through d. Rule 1 removed the other three EMSs because the origin of the item coincides with the origin of the EMS such that $x_{O_i} = x_{O_E}$, $y_{O_i} = y_{O_E}$ and $z_{O_i} = z_{O_E}$.

The second item is placed against the first one as depicted in Figure 2e. The decision regarding the exact placement is irrelevant here and will be covered in the sections below. After adding the second item to the crate, we apply Rule 3. Hence, we only update EMS 2 according to the difference process (Definition 1) as only this EMS interferes with the second item. Using the difference process on this EMS, three new EMSs are created, as shown in Figures 2f through h. EMS 4 is fully enclosed by EMS 1, thus as per Rule 2 it gets removed. The set $\mathbb{S}$ of EMSs only contains EMS 1, 3, 5 and 6. This process continues as more items are added inside the crate.



(a) Placement item 1 (b) EMS 1 (c) EMS 2 (d) EMS 3

(e) Placement item 2 (f) EMS 4 (g) EMS 5 (h) EMS 6

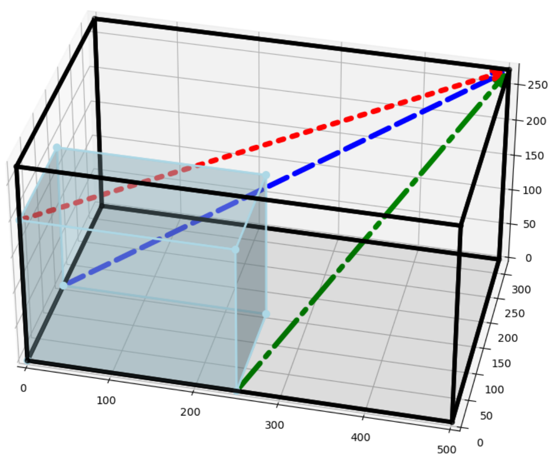
Figure 2: The creation process of EMSs after placing two subsequent items.

4.2.2.2 DFTRC-1

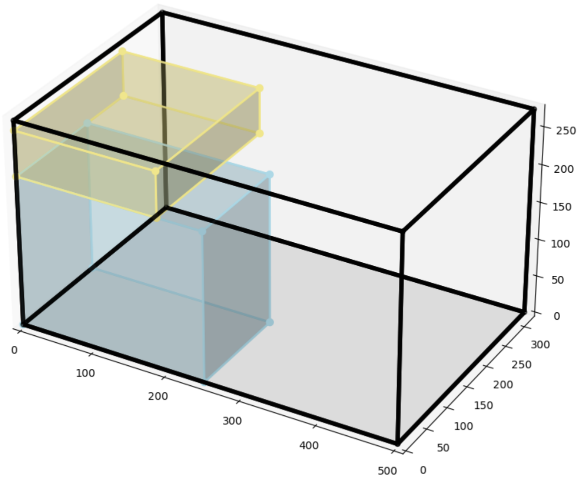
Gonçalves and Resende (2013) describe two main placement heuristics. These will be used to determine the EMS and orientation in which to place an item. The first heuristic described is 'Distance to the Front-Top-Right Corner - 1' (DFTRC-1). When placing an item, the first step is determining the feasible set of EMSs $\mathbb{S}_1$ for that item. $\mathbb{S}_1$ consists of all EMSs for which it holds that the item fits in an EMS in at least one orientation. The heuristic will only consider $\mathbb{S}_1$. For all EMSs in $\mathbb{S}_1$, the distance d from the Bottom-Back-Left corner of the EMS to the Front-Top-Right corner of the crate is computed. Figure 3 visualises the procedure, note that it uses the EMSs from the previous example in Figure 2, where again the first item is placed at the origin of the crate. Let us assume that the next item that needs to be placed fits in all of the three defined EMSs, i.e. all EMS are feasible for that item. The coloured arrows within the EMS represent the the distance d_1 per EMS in $\mathbb{S}_1$, then the EMS for which d_1 is maximised is selected for the item to be placed in. For the example, this would be the EMS of Figure 2d as the red dotted arrow has the greatest length. Subsequently, we choose the orientation based on the element of the orientation vector of the chromosome's key corresponding to that item. We use the proposed function by Gonçalves and Resende (2013) for this purpose. The heuristic keeps track of a list with feasible orientations for the EMS in which the items get placed. The actual orientation is determined based on an index from the list. To summarise, the index is determined using information of the 'orientation key' from the chromosome and the possible orientation(s) which follow(s) from the placement heuristic. The exact relation is given by the following formula:

$$Index = \lceil \text{Orientation Key} \cdot \text{Number of feasible orientations} \rceil \tag{28}$$

The advantage of using this decision rule is that only feasible orientations are considered and thus infeasible solutions are avoided. The final step of the heuristic is to place the item with its origin at the origin of the selected EMS in the chosen orientation.



(a) Selecting the EMS by maximising distance



(b) Placing the item in the selected EMS

Figure 3: The DFTRC-1 placement heuristic

4.2.2.3 DFTRC-2

The second placement heuristic (DFTRC-2) Gonçalves and Resende (2013) proposed has many similarities with the first one (DFTRC-1), but performs some of its steps in reversed order. The first step however, remains the same as in DFTRC-1, i.e. creating the list of feasible EMSs $\mathbb{S}_2$ when placing an item. Second, for each EMS in $\mathbb{S}_2$ we fix the orientation based on the chromosome's key and the feasible orientations according to Equation (28) for the selected item. Next, we virtually place the item in each of the feasible EMSs with the above determined orientation. Figure 4 shows an example of this step. Note that each coloured (red, blue and green) item is the same item but in different orientations and EMSs. We again assume that the selected item fits in all of the three defined EMSs. Let distance d_2 be defined as the distance from the Front-Top-Right corner of the virtually placed item to the Front-Top-Right corner of the crate. In the final step d_2 is maximised, with the maximum d_2 corresponding to the EMS in which the item will be placed. In the example these distances are depicted by the coloured arrows where the blue arrow has the greatest length. Therefore, this heuristic places the item in the EMS depicted in Figure 2b. Note that this differs from DFTRC-1 heuristic, while both heuristics attempt to pack crates as dense as possible.

A variation on this heuristic given by Gonçalves and Resende (2013) ignores the orientation key. They loop over all feasible orientations in each EMS and choose the orientation such that distance d_2 for each EMS is maximised. Then we place the item within the EMS corresponding to the maximum d_2. This approach would limit the strength of a GA, in the sense that you manually optimise over the orientations, while the idea of a GA is that you let 'nature' pick the best orientation. However, we use this approach within the construction of different methods without an orientation key, which is done in Section 4.4. This variation will be referred to as DFTRC-2B.

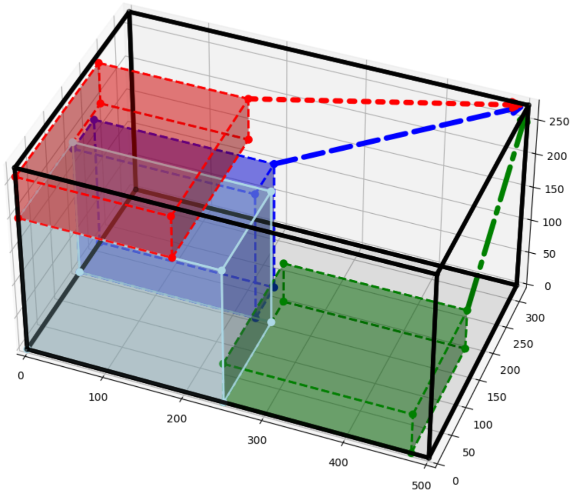


Figure 4: The DFTRC-2 placement heuristic

4.2.3 Parameters & Stop Criteria

For the BRKGA to work properly we need to make a selection of parameters. As mentioned above the BRKGA consists of a Population, and uses Selection, Crossover and Mutation operations. For each of these we need to perform a parameter evaluation to find the best possible combination of parameters to let the BRKGA perform at its best. The parameters we need to evaluate are: Population Multiplier (p_{mult}), Elite Proportion (p_e), Crossover Probability (p_c), Mutation Proportion (p_m) and Maximum number of Generations (g_{max}). The Population Multiplier determines the size of the population in each generation. For the population size we multiply the order size n with the Population Multiplier p_{mult} to get the population size. The Elite Proportion determines what proportion of chromosomes from the previous generation carried over to the next as well as the size of the pool from which the first parent (elite parent) is chosen to crossover. The Crossover Probability p_c determines the probability that a component from a vector in the offspring chromosome is from the elite parent with probability p_c and from the non-elite parent with probability $1 - p_c$. The Mutation Proportion states what proportion of the next generation is a mutant. As mentioned before, each mutant is randomly generated. Finally, the Maximum number of

Generations determines the maximum duration of the algorithm, it states the number of generations the algorithm is allowed to perform.

The algorithm also has to terminate at some point. One of the stopping criteria is therefore g_{max}. The algorithm terminates when the maximum number of generations is reached. However, some order might not need this amount of generations in order to find the best (or even optimal) solution. Therefore we introduce a lower bound (LB). If in a generation there is a chromosome with a solution equal to this lower bound the algorithm terminates as well. After both terminations, the best chromosome (determined by the fitness) is returned. The LB can be determined as follows:

$$LB = \left\lceil \max \left\{ \sum_{i=1}^{n} \frac{l_i \cdot w_i \cdot h_i}{L \cdot W \cdot H}, \sum_{i=1}^{n} \frac{g_i}{G} \right\} \right\rceil. \tag{29}$$

We thus sum over the volume per item and divided by the volume of the crate and also sum over weight per item divided by the weight limit of the crate. When taking the ceiling function over the maximum it gives us the minimum required amount of crates needed to fit all the (partial) items of the order. Therefore, when we have reached this lower bound with the algorithm we know it is impossible to fit the order in less crates and thus we can terminate the algorithm as it has found the optimal amount of needed crates. The g_{max} and LB thus form our stopping criteria.

4.3 Fragility

The above described GA aims to achieve high quality solutions to the 3D-BPP within a reasonable amount of time. However, it does not take any other practical limitations into account. This section is dedicated to one possible limitation, namely the fragility of the items. Fragility limits the placement options of items. Therefore, as an extension to the GA proposed above, we want the algorithm to avoid situations in which non-fragile (usually large and heavy) items are placed on top of fragile items. One reason to consider this method, is to prevent that fragile groceries are crushed by heavy items.

To incorporate fragility into the current problem, we first need to define the range of fragility levels that will be considered. For generality, we define q levels of fragility. The value 0 is assigned to non-fragile items. The values 1 to q are assigned to fragile items in which the value q corresponds to the highest degree of fragility. The general case encapsulates the binary fragility problem, in which items are specified to be either fragile or non-fragile. Appendix A describes the assignment of fragility to the items in the data.

In addition to the original problem, we add the following limitations to incorporate the fragility of the items. Items with a lower fragility are not allowed to be allocated above items with a with a higher fragility. In other words, from the perspective of an item, it is not allowed to come across items with a lower fragility when moving directly upwards (in the z-direction). The model based on this assumption will be referred to as the 'Standard Fragility Model'. Note that this model does not restrict items of the same fragility to be placed on top of each other.

One might argue that some products should always be placed on top and that even items with the same fragility are not allowed to be placed on top. Therefore, we will consider an 'Adjusted Fragility Model' as well. In this model we add an assumption that extends the Standard Fragility Model, which forces the most fragile items (items with fragility level q) to be on top as no other items are allowed to be placed above it.

4.3.1 EMS Update

The fragility extension strongly limits the allocation of items. Besides each item getting assigned a fragility level, each EMS should also possess information of the allowed fragility level. In order to incorporate this, we propose a method where we keep track of the minimum and maximum fragility in each EMS indicating which item types are allowed to be placed inside that EMS. This method builds upon the method described in Section 4.2.2.1 where we extend Rules 1, 3, 4 and 5 with a set of additional rules. Furthermore, we introduce an adjustment of Rule 2, such that it encloses the current extension. The rule and its adjustment are now incorporated within Definition 2.

Rule 6 describes the key adjustment to the algorithm in order to include the fragility extension. It states that EMSs are assigned a fragility level, which influence the allowance of allocating items to that EMS.

Rule 6 *If an EMS has a minimum fragility level of m and a maximum fragility level of M, with $m \leq M$, only items with a fragility level larger or equal to m and smaller or equal to M are allowed to be placed inside this EMS.*

On its turn, the placement of items influence the fragility levels of the EMSs within that crate. Therefore we make small adjustments to the difference process. Definition 1 describes the creation of all new EMSs after an item is placed inside an EMS. As follows from the Standard- and the Adjusted Fragility Model every EMS created above or beneath the item needs to be updated according to the fragility of that item. Equation (26) and (27) from the difference process are responsible for the creation of these EMSs. This means that, in terms of fragility, four out of the six EMSs created as a result of the difference process remain unchanged, while two of them need updating. Hence, the minimum fragility level of the EMS that is created above the item and the maximum fragility level of the EMS that is created below the item need to change according to the fragility of that item. Equation (26) from the difference process describes the EMS that is created below a placed item. The maximum fragility of that EMS is determined by taking the minimum of the item fragility and the maximum fragility level of the EMS taking part in the difference process (in Definition 1 described as EMS E). At the same time, the minimum fragility level within that EMS (i.e. the minimum fragility level of EMS E) remains unchanged. Similar logic is used for the EMS that is created above a placed item. Equation (27) from the difference process describes the EMS that is created above a placed item. The minimum fragility of that EMS is determined by taking the maximum of the item fragility and the minimum fragility of the EMS of taking part in the difference process (again in Definition 1 described as EMS E). This time, the maximum fragility level of the original EMS is preserved.

These two adjustments lay a first hand on preventing that non-fragile items are placed on top of fragile items, but are not sufficient to guarantee that it never happens. It can occur that EMSs are overlapping while they are defined by different fragility levels. This means that there are areas within the crate with inconsistent fragility levels. Therefore, in order to completely rule out that non-fragile items are placed above fragile items, an additional set of restrictions needs to be imposed. The proposed method for this purpose is described in Definition 2.

Definition 2 *The* **fragility update process** *is the method we apply each time an item is placed, to update the minimum and maximum fragility level of the EMSs. In this approach, all EMSs are iteratively compared to each other. Suppose we compare EMS E defined by the coordinates $[(x_{O_E}, y_{O_E}, z_{O_E}), (x_{F_E}, y_{F_E}, z_{F_E})]$ to EMS V defined by coordinates $[(x_{O_V}, y_{O_V}, z_{O_V}), (x_{F_V}, y_{F_V}, z_{F_V})]$. Three situations can occur, namely, (1) EMS E and V are non-overlapping, (2) EMS E and V partly overlap or (3) EMS E is completely enclosed by EMS V. For each of the situations we discuss the consequences.*

1. *If the two EMSs are disjoint, no updating is needed.*

2. ***IF** the top of EMS E is below the bottom of EMS V $(z_{F_E} < z_{O_V})$*
   AND

   EMS V starts to the left of EMS E $(x_{O_V} \leq x_{O_E})$ **OR** *EMS V ends to the right of EMS E $(x_{F_V} \geq x_{F_E})$*
   AND

EMS V starts in front of EMS E ($y_{O_V} \leq y_{O_E}$) **OR** *EMS V ends further back of EMS E ($y_{F_V} \geq y_{F_E}$)*

AND

the minimum fragility of EMS E is larger than the minimum fragility of EMS V,

THEN *we update EMS V in the following way:*

The minimum fragility of EMS V is set to the minimum fragility of EMS E.

3. *If EMS E is completely enclosed by EMS V, we update V by setting the minimum fragility to the maximum of the minimum fragility of E and V. After which we delete E.*

Definition 2 finalises the methods and rules necessary to comply with the 'Standard Fragility Model'. On the other hand, there are no limitations with respect to placing items with fragility q on items which also have fragility q. This creates no problem within the framework of the 'Standard Fragility Model', but it does in the 'Adjusted Fragility Model'. In order to comply with the 'Adjusted Fragility Model', one additional rule has to be composed.

Rule 7 *If an EMS has minimum fragility level equal to q, the EMS is deleted.*

Rule 7 states that when an EMS only allows items with fragility level q, the EMS can be removed. This makes sense, because in order for an EMS to have a minimum and maximum fragility of level q, there must be an item below the EMS with fragility level q. Furthermore, since the 'Adjusted Fragility Model' does not allow for allocation of items above items with the highest fragility level, we should remove this EMS.

4.4 Comparisons

The first comparison we make is between the two main solution approaches, namely solving the problem as an integer linear programming problem and the GA. These two approaches are fundamentally different as the ILP solves the problem exactly and the GA is unable to guarantee optimality. Therefore, the comparison is slightly skewed. Especially for the GA, it would be useful to have other comparison methods that are considerably quicker. Constructing adequate heuristics enables us to make a fair comparison between the heuristics and get insights in the performance of the GA relative to a 'simple and fast' heuristic. It also provides us with a better understanding of the trade-off between running time and output performance (amount of crates used). Below we will discuss two different heuristics for comparing purposes. The two heuristics are limited in their packing sequence flexibility, i.e. the order in which the items are packed. Fixing the sequence beforehand, based on logic reasoning, will put the flexibility of the GA really to the test. The first heuristic fixes the sequence in advance, while the second heuristic applies a best fit method over the remaining items. Both heuristics will use the DFTRC-2B placing heuristic as this provides the closest resemblance to the placement heuristic used in the BRKGA.

4.4.1 First Fit Descending Heuristic

The First Fit Descending Heuristic (FFD) uses a fixed sequence in which items are sorted (in descending order) based on volume. This provides us with an intuitive sequence in which the most voluminous items are placed first, since smaller items can be placed more easily in later stages. Each item is placed according to the DFTRC-2B placement heuristic into the first crate possible, much like the GA.

We expect this heuristic to be extremely quick, yet providing worse solutions compared to the BRKGA. This highlights the strength of using a good sequence, which might not be an intuitive one. Also, the heuristic has to determine the orientation according to the fixed sequence. It selects the orientation based on the 'best' orientation found by the DFTRC-2B. The GA can evolve itself over generations and acquire the knowledge that maximal distance might not entice 'best' orientation, therefore it optimises the orientation, according to the sequence, over the long run. On the other hand, the heuristic determines the orientation without this knowledge. Thus, in combination with a possible worse sequence, it might also provide an inferior orientation.

4.4.2 Best Match Densest Fit Heuristic

The Best Match Densest Fit (BMDF) heuristic does not sort items. Instead, the BMDF heuristic iterates over all available crates as well as the remaining items (the still to be placed items). For each item it determines the best position and orientation based on the DFTRC-2B heuristic. Furthermore, the crate is determined based on a ‘best match’ approach. The best match is defined as a match between a crate and an item such that, after placing the item, the fill rate of that crate is maximised. Hence, it aims to fill each crate as much as possible before opening a new crate. It is likely to place relatively large items first after which it can prefer to place smaller items such that the fill rate within a crate is maximised.

We expect this heuristic to be slower than the FFD heuristic, while still being significantly quicker than the BRKGA. Similarly to the FFD heuristic, the BMDF provides us with insights of the importance of the combination of a good sequence and adequate orientations. We expect the BMDF heuristic to provide higher quality solutions, compared to the FFD heuristic, due to the increased freedom for the determination of the sequence. However, following the same reasoning as for the FFD heuristic, we expect worse solutions compared to the BRKGA.

5 Practical Adjustments

The ILP as well as the BRKGA are expected to have a decently long running time. In order to reduce the time needed per order, we introduce different adjustments to both approaches. First, the adjustments for the ILP will be presented. Where after, the adjustments for the BRKGA will be discussed.

5.1 Exact Solution Approach

Since our solver uses a Branch and Bound algorithm, it is expected that proper bounds will improve the performance of the solver. First, we will focus on an upper bound, since this bound is necessary in the formulation from Section 4.1 in Constraints (6).

In order to obtain a fast upper bound, we introduce our No Rotation First Fit Descent (NRFFD) heuristic with local search. Since the quality of the heuristic is not the objective of this method, we mainly focus on a very fast and feasible solution. Therefore we use a FFD heuristic where rotation of the items is only allowed when they would never fit into a crate in their original dimensions. Before placement, the items are sorted in a descending order based on their volume. This assumption is based on the idea that larger items are harder to place than smaller ones.

After the NRFFD, which one can describe as a constructive heuristic, we aim to improve this solution by using a local search. This local search is constructed as follows. First, it considers the least filled crate (in terms of volume), empties it and then removes it from the solution. The next step tries to relocate all those items into the crates that still exist in the solution. If it is possible to reallocate all the items successfully, the procedure is repeated. Otherwise, the heuristic stops and returns the solution before the crate was emptied. This approach allows us to take advantage of quicker running times for smaller sized sub-orders, while trying to minimise the effects on the solution quality.

The results from the heuristic used for the upper bound can even be used more efficiently. Since a tight upper bound hardens the solver to find a feasible solution, we add a warm start to the model. The solution from the NRFFD will be used as an initial starting point from which the solver will start branching.

It is expected that the model can also be improved by adding a lower bound. That way we prevent the branching tree from unnecessarily verifying a certain solution to be optimal, by visiting lots of nodes that are infeasible. We implement the already described lower bound from Equation (29).

5.2 Genetic Algorithm

As mentioned earlier, in Section 4.2, extensive running times can occur within the GA for large orders, and/or when the chosen parameters allow for broad populations or a large number of generations. Furthermore, large orders are especially less likely to achieve their lower bound and therefore the algorithm is less likely to terminate early. It is observed that these large orders significantly influence the average running time per order. Therefore, it is desirable to decrease the running time of the algorithm for these instances. We will list multiple method for the GA, which could improve the algorithm in terms of running time. Note that although each method is explained separately, it is possible to combine them.

5.2.1 Sub-Orders

One of the stopping criteria of the GA is a lower bound based on the LP-relaxation. That means, by dividing the cumulative volume of all the items within an order by the volume of a single crate, we determine the minimum number of crates necessary when using a fill rate of 100%. Implicitly, this means that items are allowed to be spread among different crates, and hence the integrality of an item is relaxed. As a result, large orders are much less likely to hit this bound because more items are 'relaxed'. In order to decrease the running time of the GA we propose a method in which we split up large order into sub-orders. We can set a size parameter s_p to indicate the minimum order size for which orders will be split. For orders larger than s_p, approximately equally sized sub-orders are created such that the size of each sub-order is below s_p. Furthermore, combining all sub-orders must return the original order.

The advantage of this approach is that the cumulative running time of all sub-orders is, in general, much lower compared to solving the large order individually. This highlights the increased complexity when solving large orders. On the other hand, combining the solutions of the sub-orders to construct the solution of the original order will almost always result in a lower solution quality. This can be easily observed since the sum of the lower bounds of the sub-orders is always greater or equal to the initial lower bound, as the ceiling function is applied multiple times. To account for this, we suggest to use the same local search algorithm as described in Section 5.1 over the combined solution, aiming to dispose of crates with low fill rates. Since the fitness function prefers a solution with unequally filled crates over equal filled crates, it is likely that this local search can improve the solution. As the expected presence of at least one crate that is more empty than all others per sub-order.

5.2.2 Stagnation

Empirically we discovered that the populations of the GA are sometimes unable to improve themselves in terms of their best solution. This might occur when it is impossible for the solution of an order to achieve its lower bound, but also if the GA is unable to reach certain parts of the solution space and thus not attain better solution or the lower bound. The algorithm will continue to run meaningless until the final generation is reached. To prevent this we can make use of a stagnation parameter s_t. If after s_t generations no improvement has been found, we terminate the algorithm and return the best chromosome.

5.2.3 Refresh Population

As explained above, it might happen that the GA stagnates. However, instead of immediately terminating the algorithm after s_t generations, we could introduce a completely new population in order to diversify the search space. We use the refresh parameter r_p ($r_p < s_t$) for this purpose, representing the number of generations of no improvement within the population. If the GA has not found a better solution after r_p number of generations, we renew the entire population with new randomly generated chromosomes and include the best chromosome from before the renewal. This enables us to possibly escape a local minima and search through a different part of the solution space.

6 Results

In this section we will discuss the results obtained from the constructed algorithms discussed in Section 4. First we will present the performance of the exact model. Secondly, the results obtained by the BRKGA are presented. After which we will compare those results with the exact solution approach as well as the constructed heuristics. Thirdly, the practical adjustments are described and finally the results when introducing fragility are discussed. All results are obtained using the Java programming language (Oracle Corporation, 1995).

6.1 Exact Solution

The results in this subsection are computed using the solver 'CPLEX' (CPLEX ILOG IBM, 2021) on a HP Ultrabook Studio G3 with an Intel Core i7-6700 HQ (quadcore) processor and 8 GB RAM. In Table 5 an overview of these results is presented. The first column states the model used. The second column gives the total number of crates used over the 1000 orders. When no solution is found, the number of crates is set equal to the upper bound given in each model. Next, the mean time per order is given in seconds. Note that the maximum run time of each order is set to 60 seconds. In the fourth column, the mean optimality gap is given. When the model did not find a feasible solution within the time limit, the gap was set to 100%. The fifth column gives the average number of nodes evaluated per order. The second-last column gives the percentage of instances on which we can guarantee optimality. The last column gives the percentage of instances in which no feasible solution can be found within the time limit. As mentioned before, in those cases we returned the upper bound as the solution.

A variety of models is considered in this section. As a basis, we consider the model described in Section 4.1 with an upper bound that is equal to the number of items. This model is named 'Weak UB'. We define the 'Strong UB' models as the models for which we have evaluated an upper bound using a heuristic. When the solution of the heuristic is used as a starting point for the solver, we define them as 'Warm Start' models. When a Lower Bound is added to a model we include '+ LB' in the name of the model.

In general, these results support the thoughts about improved bounds and a warm start. However, when adding a tight bound, difficulties can occur in finding a feasible solution. Furthermore, the importance of adding the lower bound can be seen when these models are compared to the models which exclude a lower bound. The lower bound increases the number of cases where we can guarantee optimality. Furthermore, this contributes to an improved running time and optimality gap.

Table 5: Exact model results

Model	Crates	Avg. time (s)	Avg. gap (%)	Avg. Nodes	Solved (%)	No solution found (%)
Weak UB	5227	32.7	28.9	10345	55.6	0.8
Strong UB	2202	30.5	32.0	13594	56.8	22.3
Strong UB + LB	2184	27.0	26.9	13868	63.5	20.3
Warm Start	2196	29.63	19.6	10721	57.5	0.0
Warm Start + LB	2172	25.8	13.3	9210	65.1	0.0

6.2 Genetic Algorithm

The results in this section are computed using an 8-core MacBook Pro 2020, with 16 GB RAM and a M1 processor. Before attaining any results, the parameters for the BRKGA have to be evaluated. As mentioned in Section 4.2.3, we need to estimate the Population Multiplier p_{mult}, Elite Proportion p_e, Crossover Probability p_c, Mutation Proportion p_m and Maximum number of Generations g_{max}. Table 6 shows all the different configurations of the parameters which will be considered.

Table 6: Parameters to be evaluated for the BRKGA

Parameters	Considered values
p_{mult}	[1, 5, 10, 20, 30]
p_e	[0.05, 0.10, 0.15, 0.20, 0.25]
p_c	[0.5, 0.6, 0.7, 0.8, 0.9]
p_m	[0.0, 0.1, 0.2, 0.3, 0.4]
g_{max}	[25, 50, 75, 100, 150, 200]

To perform the parameter evaluation we make use of a stratified data sample (see Appendix B), to reduce the running time needed to estimate the best parameters. The main results from the parameter evaluations are mainly in line with the intuitive expectations. The solutions improve when using a larger population multiplier as well as allowing a greater maximum number of generations. Furthermore, the algorithm prefers a relatively high elite and low mutation proportion in combination with a low crossover probability. This indicates that slow evolution in order to attain good or near optimal solutions seems necessary. Exact results and a more in-depth view of the parameter evaluation and its results can be found in Appendix C.

From the parameter evaluation it is determined that the following parameters attain the best solution per placement heuristic for the BRKGA:

$$p_{mult} = 20 \quad \& \quad p_e = 0.25 \quad \& \quad p_c = 0.5 \quad \& \quad p_m = 0.0 \quad \& \quad g_{max} = 200 \quad \text{(DFTRC-1)}$$
$$p_{mult} = 30 \quad \& \quad p_e = 0.20 \quad \& \quad p_c = 0.5 \quad \& \quad p_m = 0.2 \quad \& \quad g_{max} = 200 \quad \text{(DFTRC-2)}$$

Table 7 displays the results obtained by the parameter settings above for the entire data set. The first column shows the used placement heuristic. Next, we give the number of crates used to pack all 1000 orders. The third column gives the average computation time per order and the fourth the median time per order. Furthermore, the total time for all orders is given in the following column.

After that, we present the average fill rate of the least filled crate, which serves as a performance measure for the placement heuristic. For this result we only take orders into account that did not reach the lower bound, because the algorithm terminates when this happens and the volume of the least filled crates is not further minimised. The average fill rate of all crates is given as well. In the second last column we show the average number of generations over all 1000 orders. Lastly, we present the number of cases where the lower bound is reached.

The table shows that the BRKGA performed best using the DFTRC-2 placement heuristic, although the average computation time was higher. Especially larger orders (orders with a quantity of at least 60 items) require the most computation time, having a significant impact on the average. This is also indicated by the median time, which is quite low. The performance of the placement heuristics can be highlighted by the average fill rate of the least filled crate. The fill rate of those crates is very low, indicating that the placement heuristics indeed first pack the first crates as densely as possible before moving on to a new crate. The algorithm also reaches the lower bound for more than 80% of the orders. The GA reaches an optimality gap of 6.8% to the lower bound, which is introduced in Section 4.2.3.

In the future sections we will use the parameter settings belonging to the DFTRC-2 placement heuristic, as it acquires the best solutions.

Table 7: Results of Genetic Algorithm

Heuristic	Nr. of crates	Avg. time (s)	Median time (s)	Total time (h)	Avg. fill rate least filled crate	Avg. fill rate	Avg. generations	Nr. of orders reached LB
DFTRC-1	1874	11.01	0.29	3.06	0.07	0.63	32	874
DFTRC-2	1866	16.78	0.36	4.66	0.08	0.64	31	882

Note: The average fill rate of the least filled crate only considers the order that did not reach the lower bound.

As mentioned above the total time in Table 7 regards the total computation time needed. This value is the average running time per order multiplied with the total number of orders. However, we would like to point out that this differs from the actual running time since the algorithm can run multiple orders in parallel. In our case, the actual running time of the algorithm was only 0.78 hours, while the total computational time is 4.66 hours. We were able to create six threads on our hardware with M1 CPU, which means that we could run at most six orders in parallel. Moreover, this implies that running time is heavily dependent on the power and the number of threads manageable by the CPU. For this reason we decided to report the computational time, although this number also depends to some extent on the used hardware.

6.3 Comparisons

We will now provide comparisons between the different proposed methods. The first comparison we can draw is between our exact approach and the GA. When comparing the results from Table 5 and Table 7, it becomes obvious that the GA outperforms the exact approach both on solution quality and computation time.

In order to further compare the results from the GA, we proposed two different heuristics in Section 4.4. Table 8 shows the results obtained by the two heuristics. The BMDF heuristic clearly outperforms the FFD heuristic. Both heuristics need considerably less computation time in comparison with the BRKGA. However, the BRKGA obtains far better results. This shows the strength of the BRKGA in terms of solution quality. When fixing the sequence of the items, as done in the FFD heuristic, it limits the possibilities and shows that the sequence is of great importance in order to obtain good solutions. When comparing the BRKGA to the BMDF heuristic, the results show that there is a more intricate relation between the sequence and orientation. It is apparently not the optimal strategy to simply select the 'best' item to be placed every time, in terms of fill rate. Hence, the evolved relation between the sequence and the orientation found by the BRKGA, is of great importance. Moreover, the comparison shows the limitation of the BRKGA as it needs time to develop and evolve this relation in the chromosomes. Thus, as the complexity of that relation grows, so does the computational time.

Table 8: Heuristic results

Heuristic	Nr. of crates	Avg. time (s)	Median time (s)	Total time (s)	Avg. fill rate least filled crate	Avg. fill rate	Nr. of orders reached LB
FFD	2345	0.00	0.00	0.72	0.19	0.51	423
BMDF	2095	0.00	0.00	0.85	0.13	0.57	654

Note: The average fill rate of the least filled crate only considers the order that did not reach the lower bound.

6.4 Practical Adjustments

In this section we present and evaluate the results of the methods proposed in Section 5, which are 'Stagnation', 'Sub-Orders' and 'Refresh Population'. The main focus of these methods is reducing the computation time without deteriorating the solution quality too much. The results are all shown in similar fashion through bar-plots. To really display the performance of the methods, we always compare them to the performance of the original GA. For all cases a computation time versus solution quality plot is created. The average computation times per order are displayed on the left y-axis, using an orange bar, and the number of used crates (solution quality) are displayed on the right y-axis, by a blue bar. Secondly, when considering methods that can directly affect the maximum number of generations used, we construct generation versus solution quality plots. These are similar to the other plot, only the left y-axis changes to the average number of generations used per order. Finally, for all plots the x-axis addresses the different parameter settings per method, where NA indicates the parameter setting of the original GA.

First, the splitting of orders into sub-orders is considered. Figure 5 shows the results of the GA when splitting orders into sub-orders and the appliance of local search afterwards. This plot also incorporates light blue bars, which represent the number of crates used before local search is applied. The darker blue bars represent the final amount of used crates. In other words, the decrease in number of crates from light to dark bars, directly represents the contribution of the local search.

As expected, a clear trade-off is seen between computation time and solution quality in terms of the number of crates. By increasing the size parameter, the solution quality increases while the computation time rises. This follows directly from the lesser amount of orders that are split-up. Non-split-up orders require more time, but also depend more on the performance of local search. The based trade-off can be found somewhere mid-way. When considering a size parameter of 35 or 40, the average computation time is at least halved and the solution quality does not suffer massively by the splits.

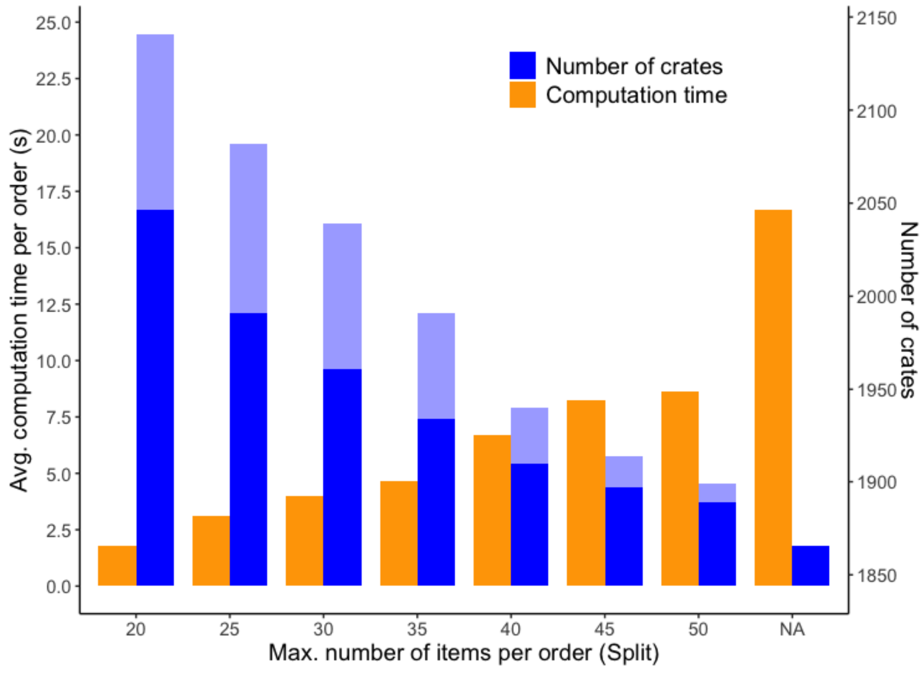
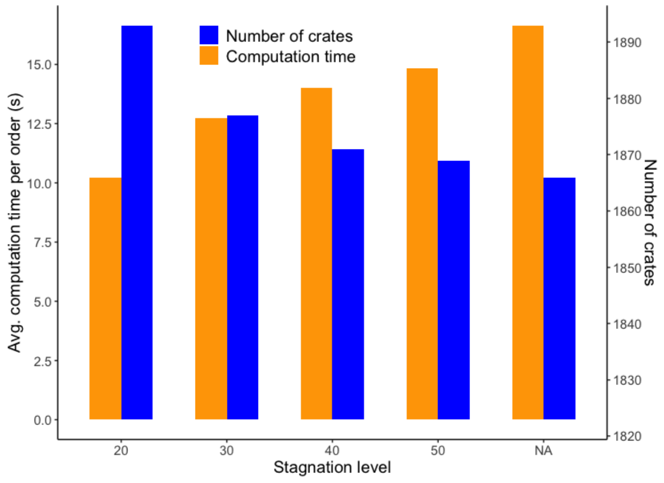


Figure 5: Sub-order results

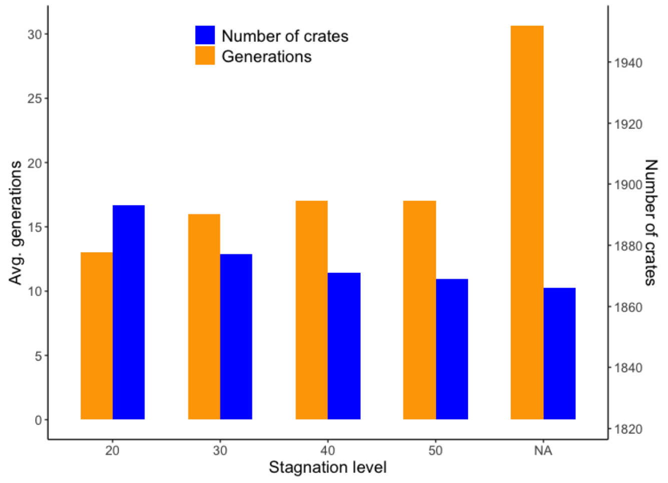
The results of our stagnation method are presented in Figure 6. The left diagram shows the average computation time versus solution quality in terms of the number of crates, whilst the right figure shows the average number of generations versus solution quality. The stagnation level s_t indicates the number of successive generations with no improvement. Thus, if after s_t generations the best fitness does not improve, the algorithm terminates.

Looking at Figure 6a, again we see a clear trade-off between computation time and solution quality. This is supported by Figure 6b, where we see that the average number of generations escalates when stagnation is not included. Stagnation does not provide the same decrease in computation time as using sub-orders does. However, it does maintain a good solution quality. When we take a stagnation level of

30, the average computation time still decreases by about four seconds and the solution quality remains nearly unchanged. Therefore, as the average generations used for any level of stagnation remains below 20, the GA should always use stagnation, since it does reduce computation time but not deteriorates the solution quality too much.



(a) Results on computation time

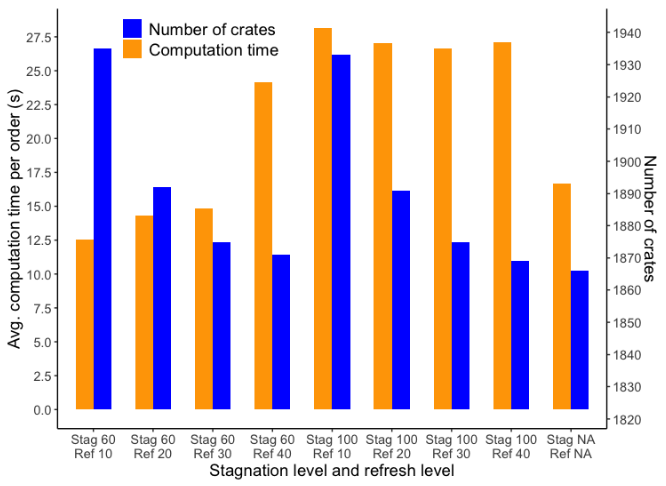


(b) Results on generations

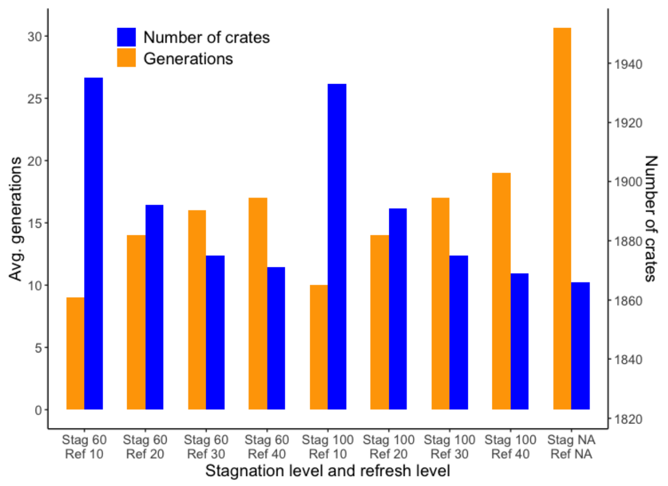
Figure 6: Stagnation results

As mentioned in Section 5.2.3 the final method, Refresh Population, should always be used in combination with stagnation. However, to ensure the proper use of this method we increase the level of stagnation compared to the levels used in Figure 6. Proper use, in this case, indicates that the population should always be able to be refreshed at least once, whilst ensuring that the population gets some time after that to evolve again. Figure 7 shows the effects of refreshing the population in combination with stagnation. Again, the left diagram shows the computation time versus solution quality and the right diagram the generations versus solution quality. On the x-axis the value of both parameters is shown.

The results show that by rapidly refreshing the population we get a significantly worse solution quality. However, refreshing the population less often significantly increases the needed computation time. When using a stagnation level of 60, it is likely that this is the only parameter responsible for the decrease in time. Therefore, it is safe to say that the algorithm probably prefers slower evolution instead of rapid refreshments of the population. Also the parameter evaluation hints at this, see Appendix B, as the algorithm prefers a low crossover probability in combination with a lower mutation proportion. We can thus quite safely conclude that refreshing the population does not yield any benefit.



(a) Results on computation time



(b) Results on generations

Figure 7: Stagnation and Refresh results

As the results from using sub-orders or a stagnation level proved to be most promising, we also evaluated a combination of the two. Figure 8 shows the results of this. It becomes clear that using this combination decreases the computation time even more compared to only using sub-orders. The addition of a stagnation level is similar to the use of only a stagnation level, in the sense that they decrease the computation time even more while not affecting the solution quality much. However, as the orders are split into sub-orders, many times the used lower bound will be achieved before the stagnation level is reached. Therefore adding stagnation contributes more if the size of the sub-orders increases. Using a size parameter s_p of 40 and a stagnation level of 40, the algorithm achieves a better solution quality with similar average computation time compared to only using the size parameter of 35. Therefore, we can conclude that a combination of these methods works very well.

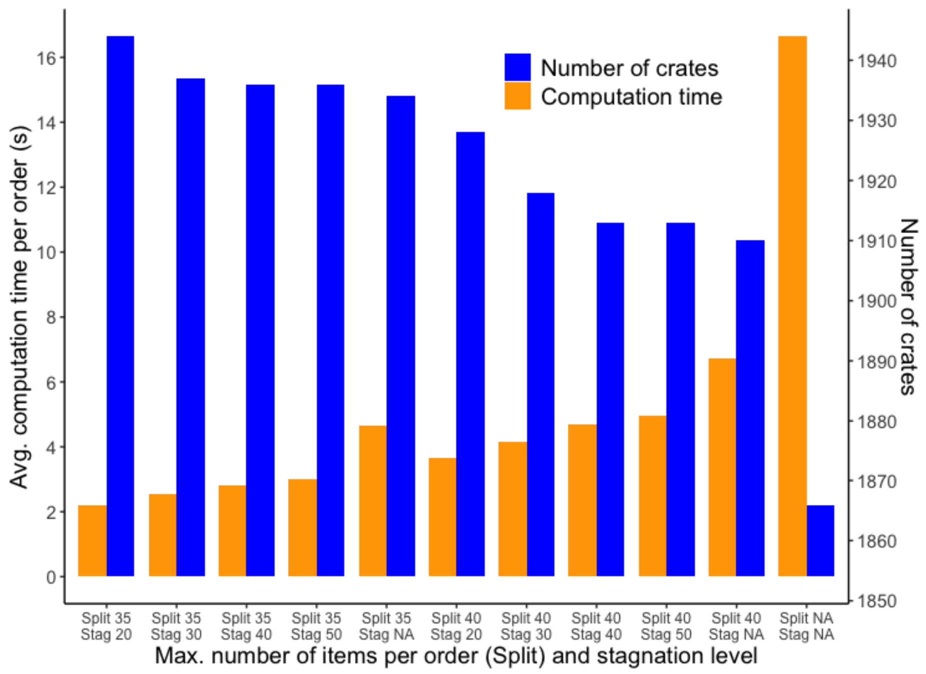
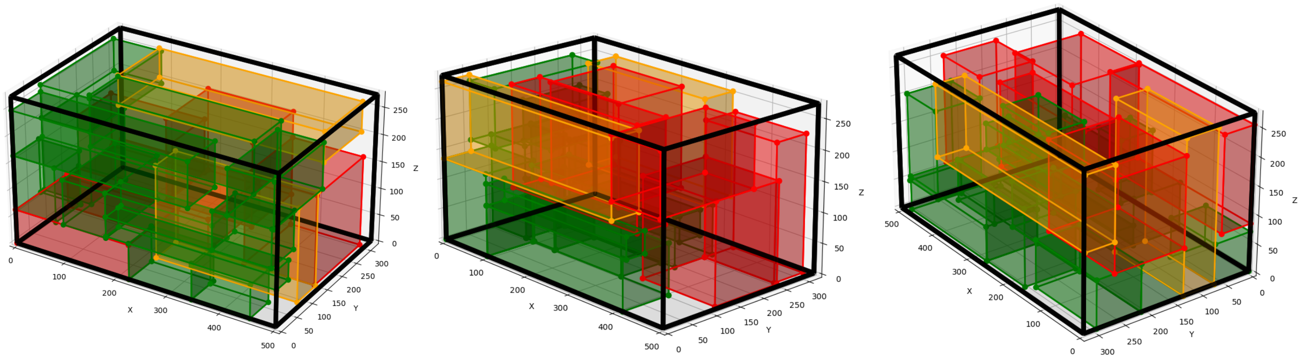


Figure 8: Split and Stagnation

6.5 Fragility

For our fragility models ('Standard' and 'Adjusted' fragility model) we consider three levels of fragility ($q = 3$). Appendix A describes the fragility level corresponding to each item. We intend to display the results of the fragility extension in the same way as the results of the BRKGA in Section 6.2. The fragility models are an extension to the BRKGA and hence the results depend on the parameter settings. Both fragility models use the same 'best' parameter setting as used in the BRKGA. Figure 9 visualises the packing configuration of order 13 resulting from the BRKGA with and without each of the fragility extensions. It is clear that the solutions follow the set of rules corresponding to each of the three different models. The remainder of this section is dedicated to the performance of each of the fragility models.



(a) BRKGA (b) Standard Fragility (c) Adjusted Fragility

Figure 9: Solution of order 13 for each of the three models. Green items correspond to fragility level 0, blue to level 1 (not present in this order), orange to level 2 and red to level 3.

6.5.1 Standard Fragility Model

As explained earlier, the Standard Fragility Model imposes restrictions on the placement of items and hence different results are obtained. Table 9 shows these results. Obviously, additional crates are needed in comparison with the results from the BRKGA without fragility extension (see Table 7), because most items are no longer allowed to be added at each possible place it fits. The total number of crates for the entire data-set is 1978, which is 112 more more compared to the BRKGA. Furthermore, the computational time per order increases with approximately 20 seconds, which is an increase of 132%. A possible explanation of this increase could be that the computation time is strongly related to the average number of generations per order. This number is namely 56, while in the BRKGA without fragility extension the average number of generations is 31. This could explain the additional computational time, while the increase of generations on its turn, could be explained by the increased difficulty of reaching the lower bound. A tighter lower bound for the fragility model is probably very beneficial for reducing the number of generations. Finally, the optimality gap to the lower bound is 13.2%, indicating that it performs worse than the BRKGA.

6.5.2 Adjusted Fragility Model

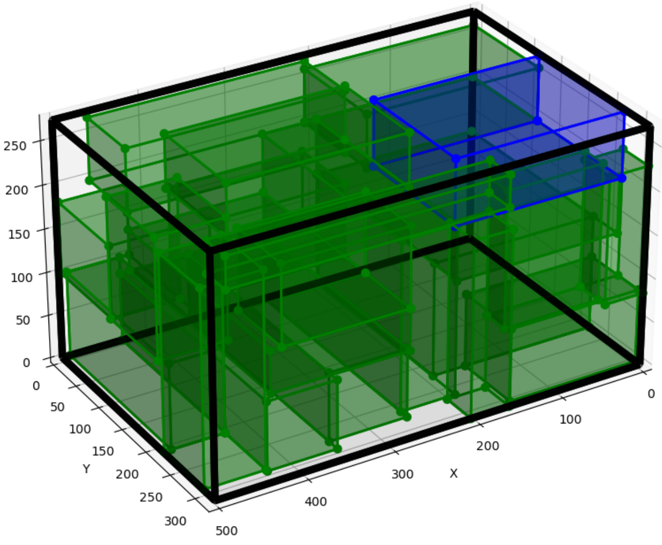
The Adjusted Fragility Model treats the most fragile items with extreme caution and its results are presented in Table 9. Since this model is the most restrictive one of the two, we would expect that this results in the highest number of crates. However, 98 additional crates are needed in comparison with the BRKGA excluding fragility extensions. This means that considerably less crates are needed compared to the 'Standard Fragility Model' (1964 versus 1978). This is counter intuitive, and we will further discuss this interesting outcome below. Also, the average computational time and average number of generations is lower compared to the Standard Fragility Model as can be seen in Table 9. Finally, the optimality gap to the lower bound for the Adjusted Fragility Model is 12.4%. This is slightly better than the Standard Fragility Model yet still considerably worse compared to the BRKGA.

Table 9: Fragility results

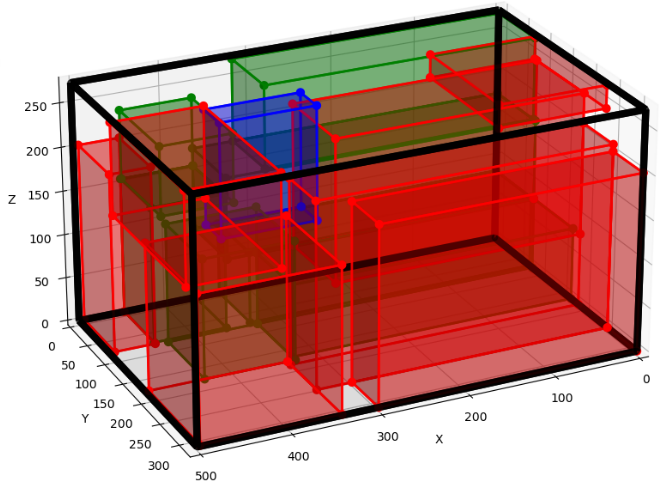
Model	Nr. of crates	Avg. time (s)	Median time (s)	Total time (h)	Avg. fill rate least filled crate	Avg. fill rate	Avg. generations	Nr. of orders reached LB
Standard	1978	38.95	0.63	10.82	0.12	0.60	56	770
Adjusted	1964	31.91	0.58	8.86	0.12	0.61	55	785

Note: The average fill rate of the least filled crate only considers the order that did not reach the lower bound.

Realising that each solution of the adjusted model is feasible in the standard model, but definitely not the other way around, made the results very unlikely at first. However, after visualising some of the orders we might have an explanation. Let us visualise the solution of order 326 of the data computed by the standard fragility model. This order fits in three crates and Figure 10 shows the first and second crate of this solution respectively. What immediately catches the eye is that crate one only consists of items of either fragility zero or one, while the second crate is mostly filled with fragile items. In other words, within the ‘Standard Fragility Model’ the genetic algorithm seems to order the items on fragility. For this to happen, initial solutions whose sequence is strongly stratified in terms of fragility must correspond to the highest fitness. We think that this is due to the fact that there are less restrictions on the placement of items, because items of approximately the same fragility are placed subsequently. Ordering the items based on their fragility may be a good selection criteria in initial generations, but this could have negative effects on the long run. In other words, the GA might be sent in the ‘wrong’ way in the solution space, such that it often gets stuck in local minima. The paradox here is that the ‘Adjusted Fragility Model’ is more restrictive than the ‘Standard Fragility Model’, but achieves higher quality solutions when using a BRKGA due to the extra restriction. Not allowing the most fragile items to be stacked upon each other makes the GA far less vulnerable to the initial bias which the Standard Fragility Model suffers from.



(a) First packed crate



(b) Second packed crate

Figure 10: Packing configuration of order 326 resulting from the Standard Fragility Model. Green items correspond to fragility level 0, blue to level 1, orange to level 2 (not present in this order) and red to level 3.

7 Conclusion

In this paper we considered the 3D-Bin Packing Problem with weight limit constraints. To solve this problem, we proposed two main solution methods. First we came up with an exact solution approach, which we tried to solve using a commercial solver. This method guarantees optimality, when the problem is solved. However, larger instances require a lot of computation time and can thus not be solved to optimality within a reasonable amount of time. It is difficult to predict which orders require a lot of computation time, however, it is strongly related to the order size. Furthermore, the performance of this method strongly depends on the provided upper and lower bounds. These drawbacks provide motivation to use a different solution approach.

Since the exact solution could not manage the complexity of the problem, we introduced the BRKGA. The BRKGA performs strongly regarding the solution quality. Furthermore, the algorithm was able to provide really fast solutions for most of the instances. However, the average computation time is still quite high due to large instances. Therefore, we introduced three adjustments to decrease the computation time while trying to maintain the solution quality. The use of sub-orders reduces the computation time the most, however, the use of stagnation maintains the best solution quality. The combination of these methods also performs really well and offers additional flexibility. The third adjustment, where the populations can be refreshed, did not show any benefits.

For the problem to be more practical, we introduced fragility constraints into the model. Each item is assigned a fragility level. The aim of the model is to prevent fragile items of being damaged. We achieved this by imposing limitations on the placement of items. This means that fragile items are only allowed to be placed on top of items that are less fragile. The genetic algorithm is particularly suitable to incorporate this extension, and we still found high quality solutions within a reasonable amount of time.

To conclude, the BRKGA is a very qualified method to solve the 3D-BPP with weight constraints. This method also offers flexibility with regard to fragility. Furthermore, based on the company's wishes it can either opt for a better solution quality or decrease the computation time by using the adjustment methods.

8 Discussion

The current paper presents two main solution approaches for the 3D-BPP. Below we will discuss some limitations and possible points of research to improve the performance of these methods.

Throughout the research it proved to be troublesome for the exact solver to close the optimality gap. Mainly it struggled with increasing the lower bound, as it needed a lot of computation time to prove that a certain lower bound was unattainable. Therefore, we expect that improving of the lower bound could significantly reduce the computation time needed and thus solve more orders to optimality.

Secondly, the current paper disregarded the practical considerations of the provided solutions. The solution quality is directly related to the fill rate. Using the solutions in practise could impose problems when crates have to be filled manually in difficult configurations. To avoid possible practical issues, we could implement a maximum fill rate.

As mentioned previously, the GA is not particularly fast. We therefore introduced multiple methods whose aim is to reduce the computation time, while minimising the negative effects on solution quality. The method in which orders are split up into subsequent sub-orders reduced the computation time the most. This can be explained by the fact that, in general, the largest orders require the most computation time. However, we cannot conclude in advance that large orders will run for a long time. Some large orders are able to achieve their lower bound within seconds and splitting would in those cases be redundant. Furthermore, with the current method, the orders are simply split-up based on index, and not on the interrelation between items or any further nuances. As a consequence, we strongly believe that the splitting technique can be improved.

A different method from which we expected that it would reduce the computation time, is refreshing the population when the algorithm does not improve for a certain number of generations. Instead, it turned out to only slow down the GA. One explanation for this behaviour could be related to the initial population and the use of random keys to represent a solution. In the creation of the initial population, randomly generated chromosomes are scattered through the entire solution space. By renewing the population, this step is repeated. The chromosomes of the refreshed population lie again scattered through the solution space. As these chromosomes are represented through random keys, it is likely that, even though the specific values of these keys differ, they represent the similar solutions. Thus not altering or moving the search space, but only repeating computations as the GA already considered these solutions and subsequently selected or rejected them. In other words, by renewing the population we think that we only slightly diversify the search space and it is unlikely to find significantly better solutions. If one would be able to always fully diversify the search space by renewing the population, it is possible that this method can attain the lower bound quicker. However, currently this is not the case.

Finally we would like to discuss that the used fitness function performs adequately, as expected. Since empirical tests showed that for the majority of orders, volume was more restrictive than weight. However, for different data this might not be the case and our fitness function could perform worse. Introducing a hybrid fitness function, that takes both volume and weight into account, could solve this problem.

Bibliography

Albert Heijn (2022). AH producten. https://www.ah.nl/producten. Accessed: 2022-02-07.

Chen, C., Lee, S.-M., and Shen, Q. (1995). An analytical model for the container loading problem. *European Journal of operational research*, 80(1):68–76.

Clautiaux, F., Dell'Amico, M., Iori, M., and Khanafer, A. (2014). Lower and upper bounds for the bin packing problem with fragile objects. *Discrete applied mathematics*, 163:73–86.

CPLEX ILOG IBM (2021). V20.1: User's manual for CPLEX. *International Business Machines Corporation.*

Crainic, T. G., Perboli, G., and Tadei, R. (2008). Extreme point-based heuristics for three-dimensional bin packing. *Informs Journal on computing*, 20(3):368–384.

Faroe, O., Pisinger, D., and Zachariasen, M. (2003). Guided local search for the three-dimensional bin-packing problem. *Informs journal on computing*, 15(3):267–283.

Gonçalves, J. F. and Resende, M. G. (2013). A biased random key genetic algorithm for 2d and 3d bin packing problems. *International Journal of Production Economics*, 145(2):500–510.

Hifi, M., Kacem, I., Nègre, S., and Wu, L. (2010). A linear programming approach for the three-dimensional bin-packing problem. *Electronic Notes in Discrete Mathematics*, 36:993–1000.

Hu, H., Zhang, X., Yan, X., Wang, L., and Xu, Y. (2017). Solving a new 3d bin packing problem with deep reinforcement learning method. *arXiv preprint arXiv:1708.05930.*

Lai, K. and Chan, J. W. (1997). Developing a simulated annealing algorithm for the cutting stock problem. *Computers & industrial engineering*, 32(1):115–127.

Liu, J., Smith, A. E., and Qian, D. (2016). The vehicle loading problem with a heterogeneous transport fleet. *Computers & Industrial Engineering*, 97:137–145.

Lodi, A., Martello, S., and Vigo, D. (2002). Heuristic algorithms for the three-dimensional bin packing problem. *European Journal of Operational Research*, 141(2):410–420.

Maarouf, W. F., Barbar, A. M., and Owayjan, M. J. (2008). A new heuristic algorithm for the 3d bin packing problem. In *Innovations and Advanced Techniques in Systems, Computing Sciences and Software Engineering*, pages 342–345. Springer.

Martello, S., Pisinger, D., and Vigo, D. (2000). The three-dimensional bin packing problem. *Operations research*, 48(2):256–267.

Mitchell, M. (1998). *An introduction to genetic algorithms.* MIT press.

Oracle Corporation (1995). *Java SE*, version 1.8. https://www.oracle.com/java/.

Parreño, F., Alvarez-Valdés, R., Oliveira, J. F., and Tamarit, J. M. (2010). A hybrid grasp/vnd algorithm for two-and three-dimensional bin packing. *Annals of Operations Research*, 179(1):203–220.

Pisinger, D. (2002). Heuristics for the container loading problem. *European journal of operational research*, 141(2):382–392.

R Core Team (2019). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria.

Snaselova, P. and Zboril, F. (2015). Genetic algorithm using theory of chaos. *Procedia computer science*, 51:316–325.

Wäscher, G., Haußner, H., and Schumann, H. (2007). An improved typology of cutting and packing problems. *European journal of operational research*, 183(3):1109–1130.

Appendix A - Fragility Selection

This Appendix describes how the fragility values are assigned to each item. Instead of defining a certain rule based on volume, weight and / or density, we selected the fragility value by hand. Since we know that the items are based on real-life items, we searched for the items on the AH website (Albert Heijn, 2022) and determined (subjectively) if they could be considered fragile or not, based on our own grocery shopping patterns. The fragility levels (FL) considered in this report are:
0: Non-fragile; **1**: Slightly fragile; **2**: Fragile; **3**: Extremely fragile.

For the selection of FL per item, we first sorted all items on decreasing weight, since the non fragile items are more likely to be heavy. Also canned, potted and boxed items, as well as bottles are considered non fragile. Extremely fragile items, are items such that the placement of a less fragile item on top of that item could cause damage. For example, bread or bags of chips are considered extremely fragile items. We rated the items of level **1** as items we would not want to put heavy items on, such as hard vegetables and fruits. And the items with level **2** are items that you would not want to get damaged, but are more likely to be put under really fragile items, such as cookies.

In Table 10 the fragility levels of the 500 items provided by AH can be found. In order to still get decent results we ensured that each level of fragility contained at least a noticeable amount of observations. Our data set consists of 39 items of level **1**, 41 items of level **2** and 48 items of level **3**. The remaining items are all considered to be non-fragile.

Table 10: Item IDs of all items with FL 1, 2 and 3

items with FL 1			items with FL 2			items with FL 3		
7	162	419	1	160	356	2	148	334
8	173	421	6	170	360	23	151	343
10	190	444	15	197	365	33	165	357
17	210	467	30	223	381	42	169	359
50	212	473	37	225	395	69	182	372
59	238	479	49	230	446	89	214	375
65	259	496	55	250	457	91	218	376
77	267		70	257	460	92	219	385
78	273		106	270	491	113	226	401
84	296		111	288		114	254	403
120	315		112	292		117	264	415
131	338		118	293		125	276	430
135	378		122	308		126	284	448
139	387		124	325		127	298	453
144	388		142	332		133	318	471
146	408		143	339		136	331	497

Appendix B - Stratified Data Sample

This Appendix covers the creation of the stratified data sample used to estimate the parameters of the BRKGA. As can be seen in Section 3.1, our data has certain properties. In order to get a valid data sample from the entire data set, the sample should be a fair representation of the original data set. First, we compute the total quantity, the total volume and the total weight per order. Subsequently we divide each of the three variables (quantity, volume and weight) into four categories. For each variable, the categories are obtained as follows:

1. Everything until the mean minus the standard deviation of the variable.
2. Everything from the mean minus the standard deviation until the mean of the variable.
3. Everything from the mean until the mean plus the standard deviation of the variable.
4. Everything from the mean plus the standard deviation of the variable.

Note that an order does not have to share the same category across the different variables.

Using the categorised variables we can perform the stratification process to obtain our sample. The sample should be a fraction of the entire data set, say f_r%. We can group orders according to their assigned categories over the variables, thus in theory creating 64 groups. However, in practice, quite a lot of groups are unattainable. In other words, it is highly unlikely for an order to be in category 1 for quantity and volume, but be in category 4 for weight. After the grouping, we can simply take the fraction f_r% of each group from the original data set to represent the new data sample. This ensures that each group is similarly represented in both the entire data set as well as in the data sample, and that the data sample has at most f_r% of the orders of the original data set.

In our case, the stratification process ensured that the stratified data sample contains at most 200 observations (f_r = 20 % of the entire data set). Due to the fact that all groups are being similarly represented both in the sample and the original data set, it is likely that the sample and the original data set have similar properties. This concludes the stratification process.

Table 11 shows the properties of the original data set and the stratified data sample. The values shown are the averages, standard deviation and median, these regard the number of items (Quantity), total volume of items (Volume) and total weight of items (Weight) per order. It can be seen that they have very similar properties and we can therefore conclude that the sample is a valid representation of the entire data set. Finally, the orders belonging to the stratified data sample are stated in Table 12.

Table 11: Properties of the original data set and the stratified data sample

	Original Data Set			Stratified Data Sample		
	Quantity	Volume (L)	Weight (kg)	Quantity	Volume (L)	Weight (kg)
Mean	26	52.22	19.73	26	51.80	19.54
Standard Deviation	13	27.91	11.08	13	27.10	10.59
1^{st} Quarter	17	32.42	11.66	16	33.83	12.09
Median	24	48.13	17.83	24	47.88	17.46
3^{rd} Quarter	33	66.53	25.81	32	65.39	25.11

Table 12: Orders in the stratified data sample

Order IDs:	1, 5, 8, 9, 10, 18, 24, 27, 28, 40 , 62 , 64 , 68 70 , 73 , 75 , 77 , 78 , 87 , 88 , 90 , 99, 103, 104, 111, 115, 119, 120, 123, 127, 133, 147, 148, 150, 155, 165, 169, 170, 173, 180, 185, 187, 189, 212, 215, 216, 217, 219, 220, 224, 234, 235, 238, 246, 253, 261, 263, 265, 269, 274, 276, 279, 293, 295, 296, 303, 311, 312, 313, 333, 336, 337, 338, 341, 343, 346, 354,, 355, 356, 359, 391, 392, 393, 397, 403, 411, 412, 413, 414, 437, 443, 449, 451, 458, 470, 472, 477, 480, 482, 483, 489, 495, 501, 502, 506, 507, 509, 518, 526, 528, 531, 532, 533, 541, 543, 552, 553, 559, 570, 574, 576, 583, 584, 588, 591, 594, 596, 597, 608, 617, 626, 631, 646, 647, 652, 654, 659, 667, 673, 694, 695, 701, 703, 711, 716, 722, 734, 737, 741, 744, 745, 749, 755, 761, 765, 768, 777, 783, 796, 797, 799, 801, 816, 820, 822, 823, 836, 848, 849, 850, 851, 869, 875, 878, 880, 886, 896, 898, 900, 905, 907, 909, 919, 920, 922, 924, 926, 932, 935, 937, 940, 942, 944, 948, 981, 982, 991, 994, 998

Appendix C - Parameter Evaluation of the BRKGA

This Appendix shows the highlights of the parameter evaluation performed on the BRKGA and discusses the main findings with regards to the stratified data sample. Table 13 shows a selection of the results obtained for the parameter evaluation. As described in Section 6.2, the parameter evaluation is performed on the stratified data sample (see Appendix B) to reduce the computation time needed to evaluate all the parameters. We found that the algorithm behaves similarly for the different placement heuristics. Therefore, to describe the main outcomes from the evaluation the choice of placement heuristic is redundant.

As expected, the solution quality increases when we increase the population multiplier or the maximum number of generations allowed. This is clearly visible in the results shown in Table 13. Furthermore, this obviously increases the average computation time per order as well.

The previous parameters behave more intuitively, however the elite and mutant proportion in combination with the crossover probability seem to have a much more delicate relation. First, when restricting the elite proportion to 0.05 and the mutation proportion to 0.5, it can be seen that increasing the crossover probability only leads to worse results. When changing the elite proportion to 0.25 and increasing the crossover probability, the results first improve a bit and then tend to get worse again. This indicates that the algorithm prefers a lower crossover probability. Thus slow evolution is preferred instead of letting the offspring immediately look very similar to the elite group. Finally, the results also show that smaller populations and smaller elite proportions prefer relatively large mutation proportions, and larger populations with larger elite proportions prefer smaller mutation proportions. In all cases, larger mutation proportions do cause larger average computation times.

In conclusion, the algorithm prefers large populations and a high number of generations, together with a relatively high elite proportion and low mutation proportion as well as a smaller crossover probability.

Table 13: Parameter evaluation highlighted results

p_{mult}	p_e	p_m	p_c	g_{max}	Placement heuristic	Nr. of crates	Avg. time (s)	Avg. generations
30	0.2	0.2	0.5	25	DFTRC-2	386	3.09	8
30	0.2	0.2	0.5	50	DFTRC-2	378	5.39	12
30	0.2	0.2	0.5	75	DFTRC-2	374	7.43	16
30	0.2	0.2	0.5	100	DFTRC-2	369	8.96	19
30	0.2	0.2	0.5	150	DFTRC-2	367	10.71	25
30	**0.2**	**0.2**	**0.5**	**200**	**DFTRC-2**	366	11.96	31
1	0.2	0.2	0.5	200	DFTRC-2	395	0.71	54
5	0.2	0.2	0.5	200	DFTRC-2	377	2.85	38
10	0.2	0.2	0.5	200	DFTRC-2	370	4.33	33
20	0.2	0.2	0.5	200	DFTRC-2	368	11.94	32
30	0.2	0.2	0.5	200	DFTRC-2	366	11.95	31
30	0.05	0.5	0.5	200	DFTRC-2	369	12.35	29
30	0.05	0.5	0.6	200	DFTRC-2	370	11.76	30
30	0.05	0.5	0.7	200	DFTRC-2	377	14.29	35
30	0.05	0.5	0.8	200	DFTRC-2	380	15.22	38
30	0.05	0.5	0.9	200	DFTRC-2	387	21.10	44
30	0.25	0.5	0.5	200	DFTRC-2	382	25.45	45
30	0.25	0.5	0.6	200	DFTRC-2	375	23.77	41
30	0.25	0.5	0.7	200	DFTRC-2	375	23.18	39
30	0.25	0.5	0.8	200	DFTRC-2	374	20.31	38
30	0.25	0.5	0.9	200	DFTRC-2	377	17.48	37
1	0.05	0	0.5	200	DFTRC-2	410	1.06	67
1	0.05	0.5	0.5	200	DFTRC-2	400	1.28	60
1	0.15	0	0.5	200	DFTRC-2	402	0.80	60
1	0.15	0.5	0.5	200	DFTRC-2	395	0.93	54
1	0.25	0	0.5	200	DFTRC-2	393	0.61	52
1	0.25	0.5	0.5	200	DFTRC-2	396	1.02	59
30	0.05	0	0.5	200	DFTRC-2	369	8.71	28
30	0.05	0.5	0.5	200	DFTRC-2	369	12.35	29
30	0.15	0	0.5	200	DFTRC-2	368	12.97	29
30	0.15	0.5	0.5	200	DFTRC-2	368	25.41	35
30	0.25	0	0.5	200	DFTRC-2	367	9.86	30
30	0.25	0.5	0.5	200	DFTRC-2	382	25.45	45