Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Systems Engineering

Bachelor Thesis

Optimization of bottle boxing using mathematical programming methods

Eder Cardoso Santana

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

BACHELOR THESIS ASSIGNMENT

Eder Cardoso Santana

Informatics

Thesis title

1. Optimization of bottle boxing using mathematical programming methods

---

Objectives of thesis

The main goal of the bachelor thesis is designing an optimal pattern of packing for bottles in order to find the best option to fit all of them in the smallest box of a pre-selected set of box sizes.
A real-world verification will be carried out with the proposed method to evaluate the results andcompare them with alternative methods found in the literature.

2. Methodology

1. The theory found in reference books and articles will be studied
2. The problem will be evaluated and all relevant data will be collected
3. The most suitable method is chosen and adapted to the problem
4. A suitable software will be chosen to solve the problem.
5. Tests will be done with the collected data and the results will be compared with reality.

3. The proposed extent of the thesis

30-40

4. Keywords

Knapsack, Packing, Optimization, Mathematical Programming

5. Recommended information sources

Everton Fernandes Silva, Túlio Angelo Machado Toffolo, Tony Wauters – Exact methods for three-dimensional cutting and packing: A comparative study concerning single container problems

Mauro Maria Baldi, Guido Perboli, RobertoTadei – The three-dimensional knapsack problem with balancing constraints

Pradeesha Ashok, Sudeshna Kolay, S.M. Meesum, Saket Saurabh – Parameterized complexity of Strip Packing and Minimum Volume Packing (2017)

Prof. Hans Kellerer, Prof. Ulrich Pferschy, Prof. David Pisinger (auth.) – Knapsack Problems-Springer-Verlag Berlin Heidelberg (2004)

Rasmus R. Amossen, David Pisinger – Multi-dimensional bin packing problems with guillotine constraints

R.S.V. Hoto, L.C. Matioli, P.S.M. Santos – A penalty algorithm for solving convex separable knapsack problems

Stefan M. Stefanov – Separable Programming: Theory and Methods (2001)

Wiley Series in Discrete Mathematics and Optimization) Laurence A. Wolsey, George L. Nemhauser – Integer and Combinatorial Optimization-Wiley-Interscience (1988)

6. Expected date of thesis defence

2021/22 SS – FEM

7. The Bachelor Thesis Supervisor

Ing. Robert Hlavatý, Ph.D.

8. Supervising department

Department of Systems Engineering

Electronic approval: 24. 11.2021

doc. Ing. Tomáš Šubrt, Ph.D.

Head of department

Electronic approval: 29. 11. 2021

Ing. Martin Pelikán, Ph.D.

Dean

Prague on 10. 03. 2023

Declaration

I declare that I have worked on my bachelor thesis titled "A Graphical Approach to Bin-Packing and Knapsack Problems in Warehouses" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break any copyrights.

In Prague on 15/03/2023 _____

Acknowledgement

I would like to express my heartfelt gratitude to my supervisor, Robert Hlavatý, for his invaluable guidance, support, and encouragement throughout my thesis. His expertise, constructive feedback, and willingness to devote his time to discuss and review my work have been instrumental in shaping this thesis.

I am also deeply grateful to my friends Gwen, Juliana, Basanta, and Nathan, who provided me with much-needed emotional support, motivation, and inspiration. Their encouragement and willingness to listen to my concerns and ideas have been crucial in keeping me on track during this challenging journey.

I would like to thank my father, Roberto, for his unwavering love, support, and encouragement. His belief in my potential and his sacrifices have been the driving force behind my pursuit of knowledge and personal growth.

I am grateful to my manager, Mostafa, for his understanding and flexibility in accommodating my academic pursuits while juggling my work responsibilities. His encouragement and support have been a source of motivation and inspiration for me.

Lastly, I want to express my deep appreciation to my partner, Ewa, for her unwavering love, support, and encouragement. Her patience, understanding, and encouragement have been crucial in helping me maintain a healthy work-life balance and a positive attitude throughout this journey.

Thank you all for your invaluable support and contributions.

Optimization of bottle boxing using mathematical programming methods

Abstract

This work proposes a solution to improve the quality of work in warehouses and tests several bin-packing algorithms and the Knapsack algorithm that guide workers to correctly pack items. As part of the study, real data was collected and modeled, based on which test instances were created for the Next Fit, First Fit, Best Fit and Worst Fit algorithms, the results of which were then processed by the DPS3UK knapsack algorithm. With the help of the obtained data, a graphic representation of optimal packaging solutions was created to help workers make better decisions. The proposed solution aims to reduce the number of accidents that occur due to poorly packed products and further optimize the use of filler materials and ensure that items are packed in the right size boxes. The study demonstrates the feasibility of this approach, although it has several limitations. However, the results offer promising potential for future improvements and applications in other industries that require efficient packaging solutions.

Keywords: Bin-packing, Knapsack, Graphical representations, Warehouse operations, Guillotine cut, Best fit, Next fit, FIrst fit, Worse fit

Optimalizace balení lahví za použití metod matematického programování

Abstrakt

Tato práce navrhuje řešení pro zlepšení kvality práce ve skladech a testuje několik bin-packing algoritmů a Knapsack algoritmus, které vedou pracovníky ke správnému balení předmětů. V rámci studie byla shromážděna a namodelována reálná data, na jejichž základě byly vytvořeny testovací instance pro algoritmy Next Fit, First Fit, Best Fit a Worst Fit, jejichž výsledky poté zpracoval algoritmus DPS3UK knapsack. S pomocí získaných dat bylo vytvořeno grafické znázornění optimálních řešení balení, které má pracovníkům pomoci lépe se rozhodovat. Cílem navrhovaného řešení je snížit počet nehod, ke kterým dochází kvůli špatně zabaleným produktům a dále optimalizovat využití výplňových materiálů a zajistit, aby byly položky zabaleny do krabic správné velikosti. Studie ukazuje proveditelnost tohoto přístupu, přestože má několik omezení. Výsledky však nabízejí slibný potenciál pro budoucí vylepšení a využití v dalších průmyslových odvětvích, která vyžadují efektivní řešení balení.

Klíčová slova: bin packing, problém batohu, grafické znázornění, skladové operace, Guillotine cut, Best Fit, Next Fit, FIrst Fit, Worse Fit

Table of content

1. Introduction

Efficient Efficient and secure storage of items in warehouses is considered essential to ensure smooth operations and reduce costs. However, packing items into boxes is a complex optimization problem that requires consideration of many factors, such as item dimensions, box sizes, and weight limits. Warehouse workers often face challenges in determining the best packing arrangements, which can lead to suboptimal packaging, wasted filler material, and even mispackaged case accidents.

To face these challenges, this thesis proposes a solution based on the application of bin-packing and DPS3UK algorithms (dynamic programming for k-staged 3UK) to test packing forms, comparing them in search of an optimal or near-optimal solution, and then create graphical representations of different packaging options that can guide workers in warehouses to pack items correctly. The main objective of this research is to find out which would be the best optimal or near-optimal packaging solutions. Creating graphical representations that can be easily interpreted and used by workers to improve their packaging decisions is a way to make the results tangible and useful for real-world use.

To achieve this goal, data is collected from a real company and modeled in a useful way for the application. An instance generator is created that can create samples for testing with collections of random items and bins. The instances are read, the packing problem is solved in four different ways to be evaluated and the results are saved. Using the results of the packing problem, a determination of packing positions within the boxes is created and saved. Finally, a graphic solution is created that demonstrates the distribution of items inside the boxes.

The contributions of this research are threefold. First, by comparing different bin-packing results, we can find which algorithms are more or less useful in different situations which can be used in future projects.

Second, a proof of concept is provided that packing problems can be graphically represented by guillotine cuts and, with sufficient improvements, can be better represented for users of these algorithms.

Third and finally, a practical solution is provided to improve the quality of work in warehouses by reducing mispackaged case accidents and helping workers make better decisions about which case to use for which collection of items.

2. Objectives and Methodology

2.1. Objectives

The main objective of the bachelor's thesis is to design an optimal packing pattern for bottles, in order to find the best option to fit all of them in the smallest box from a pre-selected set of box sizes.

A real-world verification will be performed with the proposed method to evaluate the results and compare them with alternative methods found in the literature.

By determining optimal or near-optimal packaging alternatives, graphical representations of these options will be created to help store and warehouse workers place objects in boxes. The partial objectives are::

- Collect data from boxes and items from a real company.
- Model data in a way that is useful for an application
- Create an instance generator capable of creating samples for tests and simulate real cases
- Create a program that, through known different algorithms, can read the instances and solve the packing problem and save the results
- Create a program that, through known algorithms, uses the results of the packing problem and creates a determination of positions for packing inside the boxes (knapsack) and saves the results
- Create a program that uses knapsack's results and creates a graphical solution that demonstrates the distribution of items within boxes
- Compare different results of the different methods between each other to identify which method is more usefull for an optimal solution

2.2. Methodology

The methodology proposed for solving a problem can be divided into two main parts: a theoretical part and a practical part. The steps involved in this methodology are as follows:

2.2.1. Theoretical Part:

1. Conduct a literary review based on the evaluation of books and scientific articles to identify relevant methods for solving the problem.
2. Study the existing literature to find practices and methodologies that have been developed to solve similar problems. This helps to build a solid theoretical foundation for this specific methodology.
3. Choose the most suitable method and adapt it to the problem at hand.
4. Select a suitable software that will be used to solve the problem.
5. Evaluate the problem and collect all relevant data.

2.2.2. Practical Part:

1. Use the methods found in the theoretical part to create an algorithm that will be used to develop a program that offers packaging alternatives to the user.
2. Develop a graphical demonstration of the packaging mode that helps the user visualize the impact of different packaging options.
3. Perform tests with the collected data and compare the results with reality.
4. Analyze the results of the graphical solution and the written code to determine if the goals were achieved.
5. Identify which points can be improved in the future.

By synthesizing the knowledge obtained in the theoretical part and the results of the practical part, a discussion can be formulated. This discussion will determine the efficacy of the proposed methodology in solving the problem at hand and provide suggestions for future improvements.

3. Literature Review

Bin packing and Knapsack problems are pretty known among the optimization problems in computer science, operations research, and applied mathematics. Bin packing problems aim to pack a set of objects into a minimum number of containers, while Knapsack problems seek to maximize the value of a set of items placed in a container subject to its capacity. Both problems are NP-hard, which means that no polynomial time algorithm is known to solve them optimally.

Hence, heuristic methods have been proposed to tackle these problems. According to (Wang & Chen, 2013), a heuristic refers to a computational technique that improves a candidate solution iteratively based on a given measure of quality to obtain an optimal solution. Such techniques can search through extensive spaces of potential solutions to find optimal or nearly optimal solutions at a reasonable computational cost, without relying on any specific assumptions about the problem being optimized. However, the use of heuristics cannot guarantee either the feasibility or optimality of the solution and in many cases, it is challenging to determine how close a feasible solution is to optimality.

3.1.1. Knapsack Problem

The Knapsack Problem, according to (Martello & Toth, 1990), is a well-known NP-hard problem in Combinatorial Optimization that requires maximizing an objective function while complying with a single resource constraint. Various forms of the 0-1 Knapsack Problem are considered, with regard to algorithmic techniques for the exact solution, such as relaxations, bounds, and reductions. To evaluate the effectiveness of the published algorithms, computational results are presented for comparison purposes.

In practical settings, the Knapsack Problem can be used to solve combinatorial optimization problems that involve selecting a subset of items from a larger set, while ensuring that the weight or volume limit is not exceeded. The problem can be applied to various scenarios, such as selecting items to be packed in a single container in a warehouse. The knapsack problem, being NP-hard, requires the use of efficient algorithmic techniques that can provide exact solutions or near-optimal solutions.

According to (Kellerer et al., 2004), the knapsack problem (KP) can be formally defined as follows: We are given an instance of the knapsack problem with item set N, consisting of n items j with profit p_j and weight w_j, and the capacity value c. (Usually, all

these values are taken from the positive integer numbers.) Then the objective is to select a subset of N such that the total profit of the selected items is maximized and the total weight does not exceed c. Alternatively, a knapsack problem can be formulated as a solution of the following linear integer programming formulation:

$$\begin{aligned} &\text{maximize} \sum_{j=1}^{n} p_j x_j \\ &\text{subject to} \sum_{j=1}^{n} w_j x_j \leq c \\ &x_j \in \{0,1\},\ j = 1, \dots, n. \end{aligned} \tag{1}$$

3-1.2 0-1 Knapsack Problem

In the 0-1 Knapsack problem, each item is either included or excluded from the container. The problem is solvable in polynomial time using dynamic programming, was well described by (Lau, 1986), and (Martello & Toth, 1990).

$$\begin{aligned} &\text{maximize} \sum_{j=1}^{n} c_j x_j \\ &\text{subject to} \sum_{j=1}^{n} a_j x_j b \\ &x_{ji} = 0 \text{ or } 1 (j = 1,2, \dots, n) \\ &\text{such that } a_j, b \text{ and } c_j \text{ are nonnegative numbers.} \end{aligned} \tag{2}$$

3.1.3. Bounded Knapsack

The problem we are considering, as in (3), involves filling a knapsack with a capacity of c using n given item types, where each type j has a profit p_j a weight w_j and a bound m_j on its availability. The objective of the problem is to choose a quantity x_j ,($0 < x_j < m_j$) of each item type such that the total profit of the selected items is maximized without

exceeding the weight limit "c." This optimization problem can be defined by (Pisinger, 1995) as the Bounded Knapsack Problem (BKP).

$$\begin{aligned} &\text{maximize } z = \sum_{j=1}^{n} p_j x_j \\ &\text{subject to } \sum_{j=1}^{n} w_j x_j \leq c, \\ &x_j \in \{0,1,\dots,m_j\},\ j = 1,\dots,n, \end{aligned} \tag{3}$$

3.1.4. Unbounded Knapsack

In the Unbounded Knapsack (UK) problem, according to (Hu et al., 2009), there is an unlimited supply of each item. The problem is also solvable in polynomial time using dynamic programming. Every ith type item has a value vi and a weight wi, and the knapsack has a weight-carry capacity b. Mathematically, the problem is defined as in (4).

$$\begin{aligned} &\max \sum_{j=1}^{n} v_i x_i \\ &\text{subject to } \sum_{j=1}^{n} w_i x_i \leq b \end{aligned} \tag{4}$$

With all variables as non-negative integers

3.1.5. Multidimensional Knapsack

According to (Skackauskas & Kalganova, 2022), the problem of the multidimensional knapsack involves a set of items I and knapsack K, where each item has a profit value and an N-dimensional weight that fills the knapsack. The objective is to select a set of items that maximize the total profit while ensuring that none of the knapsack capacities are exceeded. Let n be the number of items and m a number of knapsacks in the problem. (5) is the formal definition.

$$\text{maximize} \sum_{i=1}^{n} x_i \times P_i$$
$$\text{subject to} \sum_{i=1}^{n} \left(x_i \times W_{i,k}\right) \leq C_k, \forall(k) \text{ where } k \in (\mathbb{N} \leq m) \quad (5)$$

Usually the d-dimension problem KP is denoted as d-KP. (d=3, 3-KP).

3.1.6. Guillotine Cut

According to Queiroz a guillotine cut according to (ABED et al., 2015) is defined as a cut that runs parallel to one side of a container and extends all the way to the opposite side. The problem of two-dimensional guillotine cutting stock involves the application of a series of guillotine cuts, where the cuts go from one edge to the opposite edge, in order to obtain several smaller rectangles from a larger stock piece, as given by (MacLeod et al., 1993). Both cited here as well as (Queiroz et al., 2012) and (Queiroz et al., 2008) have relevant work on algortithms and lemas for working on this method.

3.1.2. Bin Packing

For(Coleman & Wang, 2013), the problem of bin-packing involves finding the minimum number of bins required to pack a given set of input data items, and it finds applications in various fields, including operations research, computer science, and engineering, where the items and bins can have diverse shapes and sizes. Since the bin-packing problem is classified as NP-hard (Garey and Johnson 1979), there is a need to develop effective heuristics that can achieve near-optimal solutions.

Using the terminology of knapsack problems, the Bin-Packing Problem (BPP) can be defined as follows: given n items and n knapsacks or bins, the task is to allocate each item to a bin such that the total weight of items in each bin is no more than c, and the number of bins used is minimized. One possible mathematical representation of this problem was made by Martelo can be expressed as in (6).

w_j = weight of item $j\big(w_j \leq c$ for $j \in N$

c = capacity of the bin (we suppose $c > 0$)

$$\text{minimize } z = \sum_{i=1}^{n} y_i$$

$$\text{subject to } \sum_{i=1}^{n} w_j x_{ij} \leq c y_i, i \in N = \{1, \dots, n\},$$

$$\sum_{i=1}^{n} x_{ij} = 1, \quad j \in N, \tag{6}$$

$$y_i = 0 \text{ or } 1, \quad i \in N,$$

$$x_{ij} = 0 \text{ or } 1, \quad i \in N, j \in N,$$

$$y_i = \begin{cases} 1 & \text{if bin } i \text{ is used;} \\ 0 & \text{otherwise ,} \end{cases}$$

$$x_{ij} = \begin{cases} 1 & \text{if item } j \text{ is assigned to bin } i; \\ 0 & \text{otherwise.} \end{cases}$$

The goal of the bin packing problem, which falls under the category of a combinatorial optimization issue, is to select the best solution from a limited number of feasible ones. A collection of goods must be packed into a predetermined number of bins in this unique instance of the knapsack problem. The bin packing problem is different from the knapsack problem in that it takes into account three dimensions of space rather than the dimensions plus one related to weight or value, and its goal is to use the fewest amount of bins possible rather than get the most value out of the goods packed.

Several methods are available to solve the bin packing problem, such as exact algorithms, heuristics, and metaheuristics. Exact algorithms such as branch and bound and dynamic programming are capable of finding an optimal solution, but they can be computationally expensive, particularly for large instances of the problem. Heuristics, such as first-fit, next-fit, and best-fit algorithms, are more efficient but may not always produce optimal solutions. Metaheuristics, such as genetic algorithms and simulated annealing, can produce good-quality solutions in a

reasonable amount of time, but they do not guarantee optimal solutions. The following methods are used to solve this problem.

3.1.3. Cutstock

According to (Haessler, 2001), manufacturers or primary converters often produce solid materials in larger sizes than needed by their customers, leading to the need to determine how to cut these materials to obtain the desired sizes. This problem is known as a cutting stock problem and can occur in one, two, or three dimensions depending on the material. The production units may vary in size, quality, and shape and the ordered sizes may also be irregular. Additionally, timing requirements may impact inventory, with some orders having different quality requirements. This method was compared with guillotine cuts by (Queiroz et al., 2012).

3.2.2. Online Methods

An online bin packing algorithm arranges items based solely on their size and the already packed items, without any knowledge of future items. Once an item is placed, the packing arrangement cannot be altered in the future, as defined in the work of (Sgall, 2014). The following are considered online methods and is reasonable to consider that First, Best, and Next fit were well analyzed by (Johnson, 1973) which uses several theorems to demonstrate how the methods work.

3.2.3. Next Fit

The NextFit (NF) algorithm, as defined by (Boyar et al., 2010), operates by maintaining only one bin at a time. If an item cannot be placed into the current bin, the bin is closed and a new bin is opened to accommodate the item. Once a bin is closed, it is not used again.

3.2.4. First Fit

The FirstFit algorithm aims to pack each item into the first available bin that can accommodate it, and if no bin is currently open that can hold the item, a new bin is opened to accommodate it, as given by (Sgall, 2014).

3.2.5. BestFit

The BestFit algorithm attempts to pack each item into the bin that has the least amount of free space but is still able to accommodate the item. If the item cannot fit into any of the currently open bins, then a new bin is opened, according to (Sgall, 2014).

3.2.6. Worst Fit

The Worst Fit WF algorithm, as defined by (Boyar et al., 2010), tries to pack the next item in a bin that has already been opened and has the minimum total size of items packed in it, if such a bin has enough capacity for the item. It opens a new bin only if there is no existing bin that can accommodate the item.

4. Practical Part

The Bin-packing and Knapsack problems are classic problems in computer science for optimization in the unused space and number of containers used. In a warehouse setting, this problem becomes more complex, as there are often constraints such as space limitations, item characteristics, and operational constraints that need to be considered. If the wrong size box is chosen during the packing process, it can lead to a number of problems that can negatively impact warehouse operations.

Firstly, if a box of the wrong size is selected, it can lead to wasted time and inefficiencies. For example, if a box that is too small is chosen, the packer may need to stop the packing process to go and find a larger box, which can waste time and reduce productivity. Similarly, if a box that is too large is chosen, the packer may need to spend additional time finding extra stuffing material to fill the empty space, which can also waste time and reduce productivity.

Secondly, choosing a box that is too large may result in unused space and stuffing. This might happen if the warehouse only has a little amount of area to work with, in which case picking a box that is bigger than necessary might lead to wasteful utilisation of that space. Also, choosing a box that is too large can call for additional filler, which would be wasteful and add extra expenses to the packing procedure.

Thirdly, selecting a box that is too small can result in damage to the box and its contents. If the box is filled beyond its capacity, it can lead to the box being damaged, which can result in items being lost or broken. This can lead to additional costs and inefficiencies, as damaged items will need to be replaced, and the packing process will need to be repeated.

Finally, selecting a box that is too tiny could not be worth the chance of improper packing of the products. If there is a chance of damage or loss, the packer may decide to take a chance and try to pack the things inside the box. This can be dangerous because even a minor error while packing can result in item loss or damage, which can be expensive and time-consuming to fix.

To address these issues, a knapsack algorithm can be used to determine the optimal way to position the items in the container after the bin-packing algorithm has been applied. The knapsack algorithm can consider the dimensions and weight (in this case the number of items in the box) of each item to ensure that the container is packed in the most efficient way

possible. This can help to reduce the risk of damage or loss of items, while also maximizing the use of space and minimizing the use of stuffing material.

In other words, bin packing and knapsack problems in a warehouse can have a significant impact on warehouse operations. Choosing the wrong size box can lead to wasted time, wasted space and stuffing material, damage to boxes and their contents, and increased risk of damage or loss of items. By using a knapsack algorithm to optimize the packing of items within the container, warehouse managers can reduce these risks and improve the efficiency of their packing processes.

4.1. The data collection

The process of data collection for this thesis was performed in person at the warehouse of the collaborating company. Due to the lack of available data regarding the items in the warehouse, it was necessary to assess each item individually to collect the necessary information for the bin packing and knapsack algorithms. The data collected included the name of the item, as well as its length, width, and height. To ensure accuracy, the measurements were taken using a digital measuring tool with an error margin of only 1 mm.

In addition to the physical dimensions, the barcode for each item was recorded during the data collection process. The barcodes play an important role in ensuring the accuracy of the packing process, as they allow for the easy tracking and identification of each item. Workers can scan the barcode to confirm that the item is in the correct location and included in the order, reducing the risk of packing errors. Furthermore, the dataset obtained from the data collection process can also be used in future studies and can be shared with other researchers to advance the field of warehouse optimization.
As part of the data collection process, I also asked the warehouse manager which types of boxes the company generally uses for packaging its products. Through this iI got to obtain a list of box models, along with their corresponding dimensions. This information was crucial for the analysis of the bin packing and knapsack problems, as it allowed me to determine the available options for packing the products.

Overall, the combination of the data obtained through the physical measurements of the products and the data of boxes provided a comprehensive understanding of the packaging

requirements and limitations of the company, which served as the foundation for the analysis and solution of the bin packing and knapsack problems.

	A	B	C	D	E	F	G
1	Nome	Code	Length	Width	Height	diameter	Weight
2	Beluga Gold Line 1,5l 40% GB	4 603 928 004 172	4,4	4,4	45,7		3062
3	Russian Standard Original 1l 40%	4 603 400 000 869			33,4	11,2	2355
4	Diplomatico Reserva Exclusiva 12y 0,7l 40%	7 594 003 620 059			21,3	8,2	1162
5	Stolichnaya vodka 0,7l 40%	4 750 021 000 065			32,7	7,2	1134
6	Don Papa 7y 0,7l 40%	4 809 015 157 015			22,5	9,5	1430
7	Capitan Bucanero Elixir Dominicano 7y 0,7l 34%	8 414 771 861 470			31,7	7,2	1282
8	Bumbu Rum 15y 0,7l 40%	813 497 006 116	9	6,3	23,5		1540
9	Legendario Elixir De Cuba 7y 0,7l 34%	8 500 000 191 026			31,5	7,6	1238
10	Jack Daniel's 5x0,05l GB	5 099 873 212 691	4,8	24,9	14,5		562
11	Proper No. Twelve 0.7l 40%	811 538 019 576			30,5	8,4	1468
12	Jack Daniel's Junkebox 0,7l 40% GB	8 595 682 604 797	10,2	16,2	29,8		1687
13	Jack Daniel's Legacy Edition 3 0,7l 43% GB L.E.	5 099 873 018 583	8	8,1	25,7		1222
14	Finlandia vodka 1l 40%	5 099 873 008 416			33	8,5	1592
15	Jack Daniel's Honey 0.7l 35%	5 099 873 001 370	7,7	7,7	24,6		1207
16	Aukce Don Papa Sherry Casks 5y 0,7l 45% L.E.	4 809 015 157 152			23,6	10,2	1557
17	Ibalgin Gin 0,7l 40%	8 595 010 604 543	6,5	8,5	23,7		1305
18	Hendrick's Gin Lunar 0,7l 43,3% L.E.	5 010 327 753 003			19,4	9,3	1226
19	Malfy Gin Original 0,7l 41%	853 222 006 189			23,3	9	1490
20	Mom Gin Rocks 0,7l 37,5%	8 410 023 095 983			22,8	9,5	1178
21	Beluga Celebration 1l 40%	4 603 928 005 476			35,6	8,6	1863
22	Finlandia vodka 0,7l 40%	6 412 709 021 776			32	7,6	1172
23	Tatratea Mini Set Mix (22-32-42-52-62-72) 6x0,(	8 588 004 169 708	3,1	24,4	11,6		714
24	Underberg 0.02l 44%	40 341 002			11	2,6	56
25	Jack Daniel's Honey 0.7l 35% + 2x sklo GB	5 099 873 204 856	9,5	19,6	26,8		2226
26	Corona Extra Pivo 11,3º 0,355l 45%	501 064 193 972			24,5	6,2	578
27	Hoegaarden Wheat Beer 0,33l 4,9%	5 410 228 141 785			21,9	6,1	545

Figure 1 – Original collected data of items, source: Own work

Krabice	(mm)	(mm)	(mm)
Poštovní krabice A5 220x150x42 3VVL 1LP	150	220	42
Kartonová krabice 400x150x150 3VVL klopová	150	400	150
Kartonová krabice 400x300x150 3VVL klopová	300	400	150
Kartonová krabice 500x300x300 3 VVL klopová	300	500	300

Figure 2 – Original collected data of boxes, source: Own work

4.2. Data Modelling

To further detail the methodology of the thesis, after the physical measurements of the products, were taken and the box models and sizes were obtained, I organized the data in a CSV (Comma-Separated Values) file. The data was arranged into columns for the name of the product, the height, width, and depth of the object, and the weight of the product in terms of units. In this particular case, each item had a weight of 1 unit, which made it easier to standardize the data.

All height, width, and depth were rounded up to the next integer value and set in cm, this gives the security that the object will fit and avoid errors as we have int values to work in the algorithms. Also, let all the measures in cm let numbers be little enough to be used in the guillotine gut algorithm since the time to run the code is related to the size of the box. This allowed for easy data processing and manipulation in the later stages of the research.

	A	B	C	D	E
1	Nome	Length	Width	Height	Weight
2	Beluga Gold Line 1,5l 40% GB	7	5	46	1
3	Russian Standard Original 1l 40%	12	12	34	1
4	Diplomatico Reserva Exclusiva 12y 0,7l 40%	9	9	22	1
5	Plantation 20th Anniversary XO 0,7l 40% GB	8	8	33	1
6	Don Papa 7y 0,7l 40%	10	10	23	1
7	Capitan Bucanero Elixir Dominicano 7y 0,7l 34%	8	8	32	1
8	Bumbu Rum 15y 0,7l 40%	9	7	24	1
9	Legendario Elixir De Cuba 7y 0,7l 34%	8	8	32	1
10	Jack Daniel's 5x0,05l GB	5	25	15	1
11	Proper No. Twelve 0.7l 40%	9	9	31	1
12	Jack Daniel's Junkebox 0,7l 40% GB	11	17	30	1
13	Jack Daniel's Legacy Edition 3 0,7l 43% GB L.E.	8	9	26	1
14	Jack Daniel's Cola 0,33l 5%	9	9	33	1
15	Jack Daniel's Honey 0.7l 35%	8	8	25	1
16	Aukce Don Papa Sherry Casks 5y 0,7l 45% L.E.	11	11	24	1
17	Ibalgin Gin 0,7l 40%	7	9	24	1
18	Hendrick's Gin Lunar 0,7l 43,3% L.E.	10	10	20	1
19	Bombay Sapphire Gin Traditional 1l 40%	9	9	24	1
20	Malfy Gin Original 0,7l 41%	10	10	23	1
21	Mom Gin Rocks 0,7l 37,5%	9	9	36	1
22	Beluga Celebration 1l 40%	8	8	32	1
23	Finlandia vodka 1l 40%	4	25	12	1
24	Stolichnaya vodka 0,7l 40%	3	3	11	1
25	Beluga 0,7l 40%	10	20	27	1
26	Russian Standard Gold vodka 1l 40%	7	7	25	1

Figure 3 – First step formatting data of items, source: Own work

	A	B	C	D	E	F
1	a,b ,c,d,e					
2	Beluga Gold Line 1,5l 40% GB,7,5,46,1					
3	Russian Standard Original 1l 40%,12,12,34,1					
4	Diplomatico Reserva Exclusiva 12y 0,7l 40%,9,9,22,1					
5	Plantation 20th Anniversary XO 0,7l 40% GB,11,11,28,1					
6	Don Papa 7y 0,7l 40%,10,10,23,1					
7	Capitan Bucanero Elixir Dominicano 7y 0,7l 34%,8,8,32,1					
8	Bumbu Rum 15y 0,7l 40%,7,9,24,1					
9	Legendario Elixir De Cuba 7y 0,7l 34%,8,8,32,1					
10	Jack Daniel's 5x0,05l GB,25,5,15,1					
11	Proper No. Twelve 0.7l 40%,9,9,31,1					
12	Jack Daniel's Junkebox 0,7l 40% GB,17,11,30,1					
13	Jack Daniel's Legacy Edition 3 0,7l 43% GB L.E.,9,8,26,1					
14	Jack Daniel's Cola 0,33l 5%,7,7,12,1					
15	Jack Daniel's Honey 0.7l 35%,8,8,25,1					
16	Aukce Don Papa Sherry Casks 5y 0,7l 45% L.E.,11,11,24,1					
17	Ibalgin Gin 0,7l 40%,9,7,24,1					
18	Hendrick's Gin Lunar 0,7l 43,3% L.E.,10,10,20,1					
19	Bombay Sapphire Gin Traditional 1l 40%,9,9,27,1					
20	Malfy Gin Original 0,7l 41%,9,9,24,1					
21	Mom Gin Rocks 0,7l 37,5%,10,10,23,1					
22	Beluga Celebration 1l 40%,9,9,36,1					
23	Finlandia vodka 1l 40%,9,9,33,1					
24	Stolichnaya vodka 0,7l 40%,8,8,33,1					
25	Beluga 0,7l 40%,8,8,35,1					
26	Russian Standard Gold vodka 1l 40%,10,10,33,1					
27	Finlandia vodka 0,7l 40%,8,8,32,1					
28	Tatratea Mini Set Mix (22-32-42-52-62-72) 6x0,04l,25,4,12,1					

Figure 4 – Final data of items, source: Own work

	A	B	C	D	E	F
1	a,b ,c,d,e					
2	PoÅ¡tovnÃ- krabice A5 220x150x42 3VVL 1LP,42,220,150,6					
3	KartonovÃ¡ krabice 400x150x150 3VVL klopovÃ¡,150,400,150,6					
4	KartonovÃ¡ krabice 400x300x150 3VVL klopovÃ¡,150,400,300,6					
5	KartonovÃ¡ krabice 500x300x300 3 VVL klopovÃ¡,300,500,300,6					

Figure 5 – Final data of boxes, source: Own work

Overall, the organization of the data in a CSV file with specific columns allowed for easy access and manipulation of the data, as well as facilitated the implementation of the bin packing and knapsack algorithms used in the analysis.

Bottles in general have irregular shapes and in order to use an algorithm to pack them, we must make the assumption that each bottle can be contained within a rectangular box with dimensions that approximate the bottle's shape. This approximation enables us to apply standard algorithms for packing rectangular items.

To further complicate matters, each bottle must be wrapped in bubble plastic to protect it during transportation. This additional packaging material must also be considered in the packing algorithm. For simplicity, I will assume that each bottle will be wrapped in bubble plastic, adding 1 cm of thickness to each side of the bottle. Therefore, when

computing the dimensions of the rectangular box that approximates the bottle, an additional 2 cm will be added to each of the three dimensions (2cm for each side).

4.3. Instance generator

A code was created that prompts the user for the number of instances to be generated and have a file path to be read, as well as a maximum number. The code then reads the specified file and selects a random number of lines from it that are less than the maximum number. The selected lines are also randomized to ensure that the generated instances are truly random.

Once the lines are selected, the code uses this information to generate multiple instances of random combinations of boxes and items. This allows for a wide range of scenarios to be simulated.

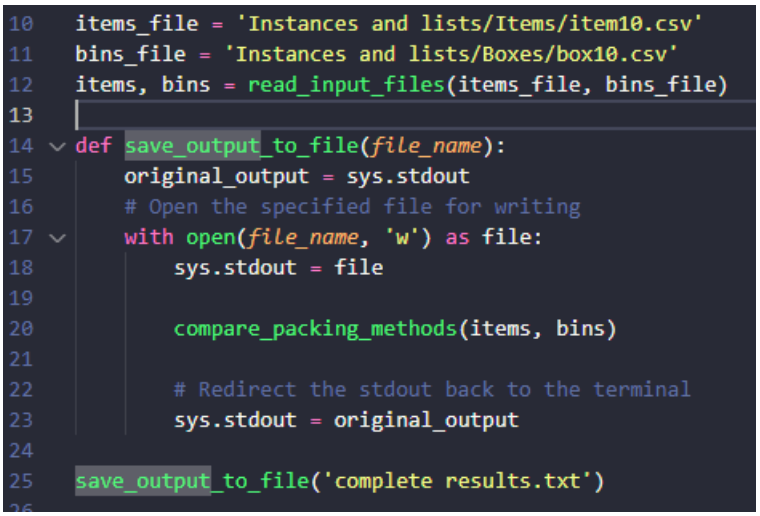
```python
import csv
import os
import random


def generate_new_csv(input_file, output_file_prefix, n, max_rows):
    # create the "Instances" folder if it doesn't exist
    os.makedirs("Instances", exist_ok=True)

    with open(input_file) as input_csv:
        reader = csv.reader(input_csv)
        headers = next(reader)

        data = list(reader)
        for i in range(n):
            num_rows = random.randint(1, max_rows)
            selected_data = random.sample(data, num_rows)
            output_file = f'Instances/{output_file_prefix}{i + 1}.csv'  # prepend the folder name to the output file path
            with open(output_file, 'w', newline='') as output_csv:
                writer = csv.writer(output_csv)
                writer.writerow(headers)
                for row in selected_data:
                    writer.writerow(row)


input_file = 'Instances and lists\All Items List.csv'
output_file_prefix = 'item'
max_rows = 10

n = int(input("Enter the number of new CSV files to generate: "))

generate_new_csv(input_file, output_file_prefix, n, max_rows)
```

Image 6 – Instance generator code, source: Own work

4.3.1. The code

A Python document named "relevant_functions.py" was created for this thesis. It contains the functions used in the main code called "main," which imports all functions and classes from an auxiliary document. The main code includes paths to instances of "items" and "bins." These paths are read and stored in the "items" and "bin" classes, respectively.

The main code also defines and uses the "save_output_to_function()" function. This function generates a file with a user-defined name and writes the output of the "compare_packing_methods()" function, which takes "items" and "bins" as parameters. "compare_packing_methods()" solves bin packing in four different ways, using the next fit, first fit, best fit, and worst fit methods. This function is based on the work of (EnzoRuiz, n.d.) , (Dube & Kanavathy, 2006), (Johnson, 1973), and (Sgall, 2014). Before each function, the start timestamp is identified, and at the end of executing the function, an end timestamp is identified to determine the time needed to carry out the process.



```
items_file = 'Instances and lists/Items/item10.csv'
bins_file = 'Instances and lists/Boxes/box10.csv'
items, bins = read_input_files(items_file, bins_file)

def save_output_to_file(file_name):
    original_output = sys.stdout
    # Open the specified file for writing
    with open(file_name, 'w') as file:
        sys.stdout = file

        compare_packing_methods(items, bins)

        # Redirect the stdout back to the terminal
        sys.stdout = original_output

save_output_to_file('complete results.txt')
```

Figure 7 – Main code, bin-pack solving: Own work

Finally, the main code prints the different possible packings found by each method. It displays the combination of items and bins, the list of items that could not be packed, and the wasted space in both absolute and relative numbers. At the end of the output, there's a brief report that includes the name of the method used, the percentage of total space wasted, and the time taken to run.

```
results > 0.txt
1  4
2  30 50 30 6
3  3 3 11
4  6 6 21
5  25 5 15
6  10 10 23
7  Kartonová krabice 500x300x300 3 VVL klopová'(30.000x50.000x30.000, max_weight:6.000) vol(45000.000)
```

Figure 8 – Result of bin-packing, source: Own work

These values can be used to understand which of the methods is most useful for the problem at hand, or even different methods can be useful in different cases of this packaging.

All methods regardless of their quirks do the following:

To store any objects that cannot be packed into the bins, the function first creates an empty list called items_not_fit.

The function adds the item to the items_not_fit_list if it doesn't fit in any of the compartments. The function calls the print results function to display the contents of each bin after all items have been processed.

The function first posts a message to indicate if anything didn't fit in the boxes before going through the list of unfit items and printing out details about each item that didn't fit.

After the results are printed, the function determines the total size of the boxes and the total amount of wasted space in the boxes by invoking the function's total wasted space and the total size of the box. After that, the function calculates the percentage of unused space for the overall size of the recycle bin and stores it in the wasted_space variable.

And finally, they use the save_output function which, for each box packed by this method, writes a .txt file with the number of items, the dimensions of the boxes, the dimensions of the items, and the name of the box. Documents are written in the cuts_results folder.

```
158     print_results(bins)
159
160     if items_not_fit:
161         print("Items that did not fit:")
162         for item in items_not_fit:
163             print(f"\t{item.name} ({item.x} x {item.y} x {item.z}) with weight
    {item.weight}")
164     wasted_space = total_wasted_space(bins)
165     tbsize=total_bins_size(bins)
166     wasted_space = wasted_space/tbsize
167     save_output(bins, 'first_fit')
168     return bins, wasted_space
```

Figure 9 – Relevant functions code, shared part in fit methods: Own work

Here is a description of what was done for each tested method:

Next fit: Each item in the items list is iterated through by the function in a try to pack it into the open bin. The item is added to the current bin if it fits. The next available bin is chosen if the item doesn't fit in the current bin, which is then marked as full. When all available bins have been used, the function stops processing items.

```
119  def next_fit(items, bins):
120      current_bin_index = 0
121      current_bin = bins[current_bin_index]
122      items_not_fit = []
123
124      for item in items:
125          if not current_bin.can_fit(item):
126              items_not_fit.append(item)
127              current_bin_index += 1
128              if current_bin_index == len(bins):
129                  break
130              current_bin = bins[current_bin_index]
131          current_bin.add_item(item)
132
133      print_results(bins)
134
135      if items_not_fit:
136          print("Items that did not fit:")
137          for item in items_not_fit:
138              print(f"\t{item.name} ({item.x} x {item.y} x {item.z}) with weight
     {item.weight}")
139      wasted_space = total_wasted_space(bins)
140      tbsize=total_bins_size(bins)
141      wasted_space = wasted_space/tbsize
142      save_output(bins, 'next_fit')
143      return bins, wasted_space
```

Figure 10 – Relevant functions code, next fit methods: Own work

Fist fit: Using a cycle, it runs through each item in the items list and tries to fit it into a bin. The function uses the can_fit method of the bin object to determine whether an item can fit into each bin in the bins list.

```python
def first_fit(items, bins):
    items_not_fit = []

    for item in items:
        bin_found = False
        for bin in bins:
            if bin.can_fit(item):
                bin.add_item(item)
                bin_found = True
                break
        if not bin_found:
            items_not_fit.append(item)

    print_results(bins)

    if items_not_fit:
        print("Items that did not fit:")
        for item in items_not_fit:
            print(f"\t{item.name} ({item.x} x {item.y} x {item.z}) with weight {item.weight}")
    wasted_space = total_wasted_space(bins)
    tbsize=total_bins_size(bins)
    wasted_space = wasted_space/tbsize
    save_output(bins, 'first_fit')
    return bins, wasted_space
```

Figure 11 – Relevant functions code, first fit methods: Own work

Best fit: The function loops through each item in the items list and tries to find the bin with the smallest remaining volume that can fit the item. If a bin is found, the item is added to the bin object. If no bin is found, the item is added to items_not_fit.

```python
def best_fit(items, bins):
    items = sorted(items, key=lambda x: x.volume(), reverse=True)
    items_not_fit = []

    for item in items:
        # Try to find the bin with the smallest remaining volume that can fit the item
        best_bin = None
        min_remaining_volume = float('inf')
        for bin in bins:
            if bin.can_fit(item):
                remaining_volume = bin.volume() - sum(item.volume() for item in bin.items)
                if remaining_volume < min_remaining_volume:
                    min_remaining_volume = remaining_volume
                    best_bin = bin

        if best_bin is None:
            items_not_fit.append(item)
        else:
            best_bin.add_item(item)

    print_results(bins)

    if items_not_fit:
        print("Items that did not fit:")
        for item in items_not_fit:
            print(f"\t{item.name} ({item.x} x {item.y} x {item.z}) with weight {item.weight}")
    wasted_space = total_wasted_space(bins)
    tbsize=total_bins_size(bins)
    wasted_space = wasted_space/tbsize
    save_output(bins, 'best_fit')
    return bins, wasted_space
```

Figure 12 – Relevant functions code, best fit methods: Own work

Worst fit: The function iterate over all items in the items list and tries to find the bin with the largest wasted space that can fit the item. If a bin is found, it adds the item to the bin object. If no bin is found, the item is added to items_not_fit.

```python
def worst_fit(items, bins):
    items = sorted(items, key=lambda x: x.volume(), reverse=True)
    items_not_fit = []

    for item in items:
        bin_found = False
        max_wasted_percentage = 0

        for bin in bins:
            if bin.can_fit(item):
                wasted_percentage = bin.volume() - sum(item.volume() for item in bin.items)
                if wasted_percentage > max_wasted_percentage:
                    max_wasted_percentage = wasted_percentage
                    current_bin = bin
                    bin_found = True

        if not bin_found:
            items_not_fit.append(item)
        else:
            current_bin.add_item(item)

    print_results(bins)

    if items_not_fit:
        print("Items that did not fit:")
        for item in items_not_fit:
            print(f"\t{item.name} ({item.x} x {item.y} x {item.z}) with weight {item.weight}")
    wasted_space = total_wasted_space(bins)
    tbsize=total_bins_size(bins)
    wasted_space = wasted_space/tbsize
    save_output(bins, 'worst_fit')
    return bins, wasted_space
```

Figure 13 – Relevant functions code, worst fit methods: Own work

For each document generated in the previous step, the code reads it and extracts the dimensions of the bin and the fitted items. These values are used as inputs for the code created by (JamesBremner, n.d.) based on the work of (Queiroz et al., 2012), which solves the DPS3UK (dynamic programming for the k-staged 3D unbounded knapsack) problem. DP3SUK.exe it is a modified version for this thesis that uses different internal funcrion and to have a single output. The code calls the executable file and passes the values as arguments.

```python
if not os.path.exists('cuts_results'):
    os.makedirs('cuts_results')

for filename in os.listdir('results'):

    if filename.endswith('.txt'):

        os.system(f'DP3SUK.exe  results/{filename} >
cuts_results/{filename}')
# vectors to store the cuts
```

Figure 14 – Main code, Solver of guillotine cut, source: Own work

The output generated by DP3SUK.exe consists of the guillotine cuts of the bin used in the packing process in all the different dimensions. This solution includes the number of cuts in each direction, and the positions of the cuts in that direction, if there are any because there's the possibility of no cuts being needed in some direction used. Also, the code provides the dimensions of the box for the next step.

After obtaining the solution for the DPS3UK problem, the code generates a new document for each bin used in the packing process. Each document contains the packing sequence for that bin, with the boxes and items packed inside them. These documents are saved in the same folder as the previous step, and their names correspond to the names of the original documents generated in the first step(properly enumerated).

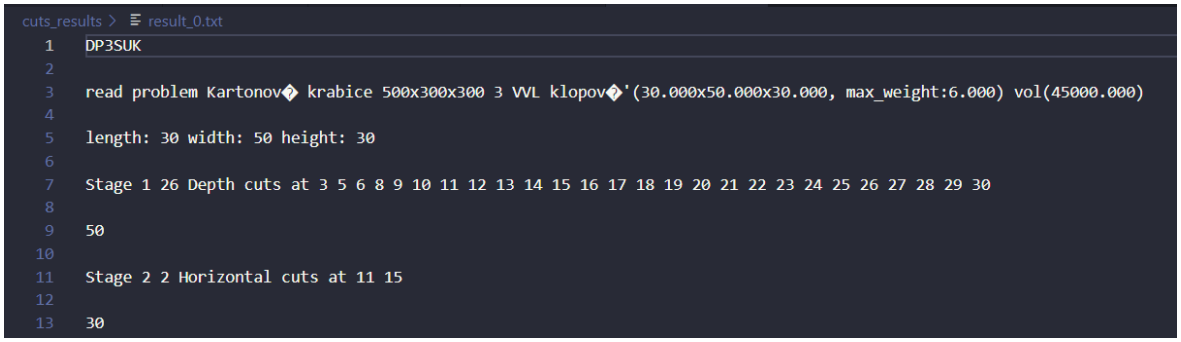


Figure 15 – Results of Guillotine cut, source: Own work.

This program differs from the original in that it uses the Reduced Raster Points (RRP) algorithm as presented by (Scheithauer & Terno, 1996) and used by (Birgin et al., 2010), (de Almeida Cunha et al., 2020), and (Kartak & Ripatti, 2018, p. 34) rather than the Discretization using Dynamic Programming (DDP) algorithm, as implemented by (Cintra et al., 2008). The DDP algorithm is used to evaluate each possible position for the cuts and verify if the items can be packed. This is guaranteed by the previous code, but the code must determine what rotation to put inside the box.

```
std::vector<int> DDP(
    int D,
    const std::vector<int> &d)
{

    std::vector<int> P; // calculated discretization points

    std::vector<int> c(D + 1);

    for (int i = 0; i < (int)d.size(); i++)
    {
        for (int j = d[i]; j <= D; j++)
        {
            if (c[j] < c[j - d[i]] + d[i])
                c[j] = c[j - d[i]] + d[i];
        }
    }
    for (int j = 1; j <= D; j++)
    {
        if (c[j] == j)
            P.push_back(j);
    }
    return P;
}
```

Figure 16 – Main code, code RRP, source: James Bremner code

After each possible position is checked in each direction, a cut is made if needed. The items that can fit inside the region delimited by that cut are stored, and the other items are evaluated considering the previous cut.

Next, the Python code reads the "./cuts_results" file and creates an array to store strings containing the names of the dimensions and the one-dimensional coordinates of each cut. Then, the program generates three planes perpendicular to each other, based on the dimensions of the bin, which are used as regions for the cuts in the x and y axes of each plane.

```
Vert = [0]
Hori = [0]
Dep = [0]
cut_vec_list = [Vert, Hori, Dep]
makes_images()
```

Figure 17 – Main code, Image generator, source: Own work

The use of only three planes is sufficient, as the cuts made on one side of the box are mirrored on the opposite side, in the opposite direction. However, at least three planes are

required, as the cuts that pass through one plane will never reach the other two perpendicular planes.

To create images, relevant lines and positions for cuts are read, such as the dimensions of the box and positions of the cuts.

```
def makes_images():
    for file_name in os.listdir(directory_path):
        if file_name.startswith("result_") and file_name.endswith(".txt"):
            file_path = os.path.join(directory_path, file_name)
            lines = read_file_lines(file_path)
            new_lines = []
            for line_list in lines:
                if line_list[0] == '':
                    continue
                new_lines.append(line_list)

            print(new_lines)

    for i, line_list in enumerate(new_lines):
        if len(line_list) > 2:
            try:
                n = int(line_list[2])
                if line_list[3] == "Depth":
                    for j in range(n):
                        Dep.append(line_list[j + 6])
                elif line_list[3] == "Horizontal":
                    for j in range(n):
                        Hori.append(line_list[j + 6])
                elif line_list[3] == "Vertical":
                    for j in range(n):
                        Vert.append(line_list[j + 6])
```

Figure 18 – Main code, function for image organization part 1, source: Own work

These are then restructured into a matrix called cut_group, which includes dimensions, cuts, and box dimensions for each dimension.

```
Vert.append(new_lines[2][5])
Hori.append(new_lines[2][3])
Dep.append(new_lines[2][1])
print("Dep:", Dep)
print("Hori:", Hori)
print("Vert:", Vert)
cut_group = [Dep, Vert, Hori]
print(cut_group)
```

Figure 19 – Main code, function for image organization part 2, source: Own work

The cut_group is passed to the cut_draws function, which iterates over the rows and columns to build an (x,y) coordinate system for a plane.

```
def cut_draws(result_number, cut_group):
    width = 1000
    height = 1000

    for i in range(len(cut_group)):
        for j in range(len(cut_group)):
            if i != j:
                turtle.clearscreen()
                print(cut_group[i], cut_group[j])

                def draw(t): return cut_in_directions(
                    cut_group[i], cut_group[j], t, width, height)

                write_file(draw, "image_{0}_{1}.svg".format(
                    result_number, i), width, height)
```

Figure 20 – Main code, function to indicate directions of images, source: Own work

Using the write_file function, an SVG document is created with an image generated by the draw function, which receives iteration coordinates.

```
def write_file(draw_func, filename, width, height):
    folder_name = create_folder()
    file_path = os.path.join(folder_name, filename)
    t = SvgTurtle(width, height)
    draw_func(t)
    t.save_as(file_path)
```

Figure 21 – Main code, function for saving SVG file, source: Own work

The draw function returns a use of the cut_in_directions function, passing the coordinates and a scale_factor (a constant that corrects the image size for initial viewing). The cut_in_directions function is used to call cuts_in_x, cuts_in_y, and draw_borders.

```
def cut_in_directions(x, y, t, width, height):

    scale_factor = 10  # min(width, height) / max(image_width,
image_height)*2

    draw_borders(x, y, t, scale_factor, width, height)
    cuts_in_x(x, t, scale_factor, width, height)
    cuts_in_y(y, t, scale_factor, width, height)
```

Figure 22 – Main code, call different image generations , source: Own work

Draw_borders creates the borders of the boxes, taking the largest x and y coordinates to create a straight line from (0,0) to (x0,0) and (0,y0), and from (x0,0) and (0,y0) to (x0,y0).

```
def draw_borders(x, y, t, scale_factor):
    last_x = int(x[-1]) * scale_factor
    last_y = int(y[-1]) * scale_factor

    t.penup()
    t.goto(0, 0)
    t.pendown()
    t.goto(0, last_y)

    t.penup()
    t.goto(0, last_y)
    t.pendown()
    t.goto(last_x, last_y)

    t.penup()
    t.goto(last_x, last_y)
    t.pendown()
    t.goto(last_x, 0)

    t.penup()
    t.goto(last_x, 0)
    t.pendown()
    t.goto(0, 0)
```

Figure 23 – Main code, border designers, source: Own work

Cuts_in_x creates straight lines between positions on the x-axis and goes to the same position in x, but at the maximum value in y. Cuts_in_y works the same way, replacing x with y. All three functions write the coordinate where the cut was made at the end point of the cut.

```python
def cuts_in_x(vector, t, scale_factor):
    for i in vector:
        x = int(i) * scale_factor
        y0 = 0
        y1 = int(vector[-1]) * scale_factor

        t.penup()
        t.goto(x, y0)
        t.pendown()
        t.goto(x, y1)
        t.write(int(i))
```

```python
def cuts_in_y(vector, t, scale_factor):
    for i in vector:
        x0 = 0
        x1 = int(vector[-1]) * scale_factor
        y = int(i) * scale_factor

        t.penup()
        t.goto(x0, y)
        t.pendown()
        t.goto(x1, y)
        t.write(int(i))
```

Figures 24 and 25 – Main code, cut plotters on the x and y axes, source: Own work

The final step of the program is to generate an HTML document that includes images of the packed bins and their corresponding names. This guide serves as a visual representation of how the items should be distributed and arranged for efficient packing. The images show the bin from the respective side, along with the cuts and sections where items have been placed. Users can refer to this guide to help them with their packing needs and to understand how the items are arranged inside the bin.

```python
def generate_html_with_svg_files():
    # Define the folder containing the SVG files
    folder = ".\cuts_of_order"

    # Get a list of all the SVG files in the folder
    svg_files = [f for f in os.listdir(folder) if f.endswith(".svg")]

    # Create the HTML document
    html = "<html><head><title>Packing options</title></head><body>"

    # Loop through each SVG file and add it to the HTML document
    for svg_file in svg_files:
        # Get the name of the SVG file without the file extension
        name = os.path.splitext(svg_file)[0]

        # Add the name of the SVG file to the HTML document
        html += f"<h1>{name}</h1>"

        # Add the SVG file to the HTML document
        html += f'<object type="image/svg+xml" data="{folder}/{svg_file}"></object>'

    # Close the HTML document
    html += "</body></html>"

    # Write the HTML document to a file
    with open("packing_options.html", "w") as f:
        f.write(html)
```

Figure 26 – Main code, HTML generator part, source: Own work

5. Results and Discussion

5.1. Results

To assess the program's effectiveness, ten randomly generated instances were tested to determine whether the program could identify alternatives in a viable time frame. Test results demonstrate that the program can be used effectively, with an average total processing time of 4.56 seconds, which is adequate for most operations. This suggests that the program can be a useful tool for employees to select the appropriate packaging alternatives for their items, thereby reducing wasted time and improving efficiency.

	A	B	C	D	E	F
1	Method	Waste (%)		Execution Time (s)	AVG	total seconds
2	----------					
3	Next Fit	85,91%	75,70%	0,001	0,00107	0,568264484
4	First Fit	74,43%		0,001		
5	Best Fit	71,23%		0,00114		
6	Worst Fit	71,23%		0,00114		
7						
8	Next Fit	98,50%	96,26%	0,001	0,001053	6,355749846
9	First Fit	97,01%		0,00119		
10	Best Fit	95,51%		0,00101		
11	Worst Fit	94,01%		0,00101		
12						
13	Next Fit	72,49%	72,49%	0,00101	0,001	0,503208876
14	First Fit	72,49%		0,00099		
15	Best Fit	72,49%		0,001		
16	Worst Fit	72,49%		0,001		

Figure 28 – Analysis of the comparison of results, source: Own work

As for the different methods used, averages of execution time and the percentage of space of the total boxes to be wasted were taken. As we can see in the following figure.

Method	Numeber of less waste	Number of over AVG waste	Number of best time	Number of over AVG time
Next Fit	2	8	5	5
First Fit	4	5	5	5
Best Fit	8	0	5	3
Worst Fit	10	0	5	3

Table 1 – Analysis of general results, source: Own work

Thus, we used a ranking method for decision making of better algorithms where the values are ordered from most efficient to least efficient:

Method	Numeber of less waste	Number of over AVG waste	Number of best time	Number of over AVG time
Next Fit	4	3	1	2
First Fit	3	2	1	2
Best Fit	2	1	1	1
Worst Fit	1	1	1	1

Table 1 – Decision over analysis of general results, source: Own work

Each method had the best execution result half of the time, next fit and first fit, when they did not have this result, were above the average execution time, different from the other two. In the end, this criterion was considerably relevant since most of the time spent depends on the guillotine cutting algorithm and the execution time is low enough to not be considered in most cases (cases where this becomes relevant will be described later), therefore the values of wasted space as more relevant.

In this way, we identified that the best algorithm was Worst fit in all criteria, Best fit is close enough to be considered in future studies. First fit and Next fit proved to be considerably less efficient.

The program's HTML output displays the instance name, along with three images representing the walls of the box from different angles. The images can be interpreted by examining the coordinates of the slices represented on the edges, with coordinate 0 coinciding with all planes. The cutpoints serve as the maximum limits that the items must

reach from the previous cutpoints. Thus, the images provide a clear and intuitive representation of the packaging alternatives available for each item.

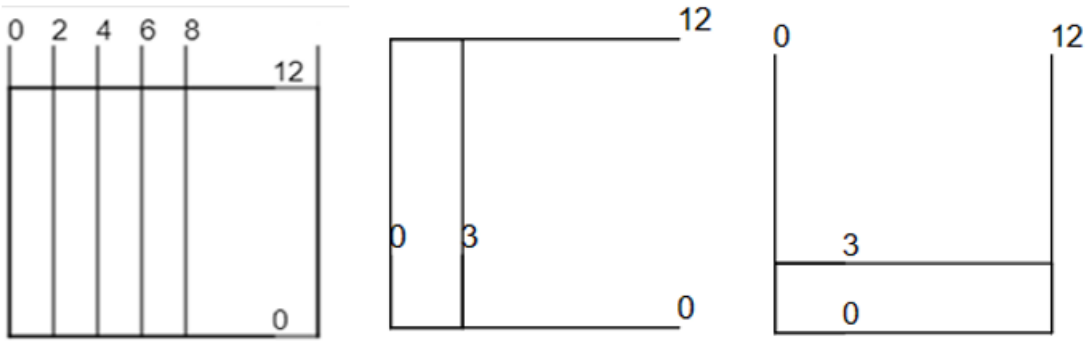


Figure 17 – Final result of cuts in one view, source: Own work

In the view of Figure 17, each slice reveals a region where an item would be seen having one of its edges. In this case we would have an item appearing horizontally between 8-14, one between 6-8, one between 4-6, one between 2-4 and a 0-2, and one vertically 0-12. Note that when receiving the image we do not know what the item is and in what relative position it is.

In addition, the study demonstrates that the program's ability to offer several packaging alternatives can guide employees in their momentary preferences, streamlining the decision-making process. By offering a range of alternatives, the program allows employees to make informed decisions based on their specific needs and preferences.

Overall, the results of this study indicate that the developed program is a valuable tool to identify suitable packaging alternatives in a timely and efficient manner, and can be improved to provide even better performance. The program's ability to guide employees through their momentary preferences demonstrates its potential to improve operational efficiency and reduce wasted time in the workplace.

5.2. Discussion

In order to fully comprehend the possible impact that this thesis could have, it is important to analyze how close the results are to an ideal solution for the presented problem. By doing so, we can identify the pros and cons of this work, both in general and for the specific warehouse packing problem.

5.2.1. Pros

First and foremost, one of the main advantages of the working code is that it was able to process the instances in an average time of 4.56 seconds. This is a relatively short amount of time, considering that it is sufficient for creating the solutions and indicating to the packagers what the optimal packing strategy would be. If the packagers were to make these decisions themselves, it could take much longer and could result in errors. This time is reasonable even for large orders of up to 10 items.

Furthermore, the program is modular in design, which means that it can be easily adapted in case of changing box sizes or if a box is missing. The program is not directly linked to a list of items, which means that there are no restrictions on item names or sizes. Additionally, modularity allows the user to change the number of items allowed per box, providing the user with flexibility.

The program shows a list of all the boxes, their corresponding names, and all the items that should go within them, along with a list of those items' names. The application furthermore offers a separate list of objects that do not fit into any of the boxes. Users can considerably reduce packing problems because to this functionality. The program provides the necessary box size effectively, saving the user time and effort while recognizing the items, even though it does not specify the exact placements of each item within the box.

Another key advantage of the generated images is that they are relevant for showing the fitting delimitations of the items inside the bins. As can be seen in drawings of Figure 17 each image represents a plane indicating the walls of the box. This allows users to identify in which region an item must be placed and what its direction should be, which is particularly useful for items that have rotations or other specific orientations.

Finally, to further aid in the visualization of the packing strategy, the numbers in the images indicate the distance of the cuts from the origin. This information enables users to understand the order of the items in terms of size.

Of course, there are also some limitations to this approach that must be considered. Despite these limitations, the results of this thesis are promising and demonstrate that the various packaging alternatives, given the possibilities, can guide employees through their momentary preferences. By implementing the modular program presented in this work, companies could potentially save time and resources by optimizing their packing strategies,

particularly for small to medium-sized orders. Further improvements to the program could make it even more efficient and adaptable for a wider range of scenarios.

5.2.2. Cons

The DPS3UK algorithm presented in the thesis has some limitations and considerations that must be taken into account before implementing it. Firstly, if the boxes used in the algorithm are considerably large, it is important to ensure that the items being packed are proportionately large as well and that there are enough items to fill the box. This is because the algorithm relies on the resulting values of DDP and the dimensions of the box. In cases where there is plenty of space in the box and it is not necessary to evaluate all viable positions, adjusting the dimensions of the boxes and items to be proportionate will suffice.

Secondly, it's worth noting that the processing time for the DPS3UK algorithm is fast enough for cases of 1 or 2 small or medium items per order. However, in such cases, it may not be necessary to use the method presented in the thesis. These types of cases are relatively trivial, and it is not necessary to use the algorithm in all situations to save computational time. Instead, the method should be reserved for more complex situations, such as those where the number of items is greater than 2, which was found in about half of the requests observed in the company.

Thirdly, the program only generates solutions based on the available boxes and item sizes. It does not take into account the overall efficiency of the packing strategy or other constraints, such as weight or fragility. Additionally, the program may not always generate the most optimal solution, particularly when dealing with a larger number of items.

Therefore, while the DPS3UK algorithm can be an effective tool for optimizing the packing process, it's important to consider the limitations and best practices for implementation. By ensuring that the dimensions of the boxes and items are proportionate and only using the algorithm when necessary, it can be a valuable tool for companies looking to streamline their packing processes and improve efficiency.

The method presented in the thesis for packaging items has some limitations and areas for improvement that need to be addressed before it can be effectively implemented. A big problem is that the images used in the method do not support writing the names of the items, which can make them more confusing, especially if there is a lot of content. Also, depending on the number of cuts made, if the writing is between the cuts, it may not be able

to be read. This leaves the method dependent on the generated images, which have restricted variations, and users may have difficulty identifying the position of the boxes.

Another challenge of the method is that there is an ambiguity of order between the objects. It is difficult to determine whether item A comes before item B or vice versa, especially if there is item C that has similar dimensions to A or B when rotated in a certain direction. For example, if A=(10, 11, 20), B=(10, 17, 23) and C=(9, 17, 20), it would be difficult to distinguish the order of items A and B. Although the method is useful and reasonable, requires a significant number of changes to overcome these issues.

It may happen that programs indicate an item as not being able to be packed because the previous item may have been placed in a position that makes it impossible for the current item to be unpacked, but none of the items are changed from position once they are packed in all the algorithms used.

6. Conclusion

6.1. Final Summary

In summary, this thesis work demonstrates the successful creation of a program to solve a specific warehouse packaging problem. The program is modular and adaptable, allowing for changes in box sizes and the number of items per box. It generates graphical solutions in a timely manner, showing the delimitation of items and coordinates of cuts. This allows employees to evaluate which boxes to use and avoid wasting time, and the generated packaging alternatives can guide employees in their momentary preferences.

However, the packaging method presented some limitations and areas for improvement in the text. The algorithm used depends on the dimensions of the box and the item and is best suited for complex situations. The method relies on images which can be confusing, has complex but unoptimized code, and doesn't show how much internal space has been affected. Despite these restrictions, the program is effective for large orders of up to 10 items and has the potential to be improved for use in warehouses with more complex situations.

6.2. Recommendations for future research

The thesis analyzed in this context focused on situations that were solved by a bin-packing method, found more useful methods for the specific job, but it's not perfectly optimal yet. As a next step, it is suggested that the algorithm be able to iterate over items even after being packed to ensure that the items can be packed, reducing wasted space.

For future work, focus should be placed on finding solutions for packaging that consume less space, without worrying about the execution time that, even running unnecessary functions for a real situation (such as writing and reading files, for example), the tests were executed quickly enough.

Additionally, the solution can be further improved by using colors and hatching to indicate which items are contained within each cut, in order to avoid ambiguity. It is also recommended to represent the layers by identifying which item should be placed first in the box (if order matters) and which direction the item must be oriented inside the box.

Furthermore, having a representation of the test plans with the cuts on the sides of the box combined in the origin would be very useful for spatial identification and understanding of how the cuts represent the items.

Ideally, an interactive version of the method could also be created, where users can select the regions inside the box to identify which item should be placed and in which direction.

Finally, a change that would make the code more flexible to individual issues would be to ensure a readjustment of box proportions in case the boxes are considerably larger than the items or be able to choose how many units of each box can be used. This would allow the method to be applied to a wider range of scenarios and improve its overall utility.

In summary, the passage highlights various modifications that can be made to improve the use and scalability of a specific method. These changes include ordering box options to minimize wasted space, using colors and hatching to avoid ambiguity, creating an interactive version of the method, providing a spatial representation of the cuts, and ensuring flexibility in adjusting box proportions..

7. References

1. ABED, Fidaa, Parinya CHALERMSOOK, José CORREA, Andreas KARRENBAUER, Pablo PÉREZ-LANTERO, José A. SOTO, Andreas WIESE. (2015). On Guillotine Cutting Sequences. In N. GARG (Ed.), Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques. (pp. 1-19). Wadern: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-89-7. Available under: doi: 10.4230/LIPIcs.APPROX-RANDOM.2015.1
2. Birgin EG, Lobato RD, Morabito R. (2010). An effective recursive partitioning approach for the packing of identical rectangles in a rectangle. Journal of the Operational Research Society, 61, 306-322.
3. Boyar, J., Epstein, L., & Levin, A. (2010). Tight results for Next Fit and Worst Fit with resource augmentation. Theoretical Computer Science, 411(26-28), 2572-2580. ISSN 0304-3975. https://doi.org/10.1016/j.tcs.2010.03.019.
4. Cintra, G. F., Miyazawa, F. K., Wakabayashi, Y., & Xavier, E. C. (2008). Algorithms for two-dimensional cutting stock and strip packing problems using dynamic programming and column generation. European Journal of Operational Research, 191(1), 61-85. Available at: https://doi.org/10.1016/J.EJOR.2007.08.007
5. Coleman, N., Wang, P. (2013). Bin-Packing. In: Gass, S.I., Fu, M.C. (eds) Encyclopedia of Operations Research and Management Science. Springer, Boston, MA. Available at: https://doi.org/10.1007/978-1-4419-1153-7_75
6. de Almeida Cunha, J.G., de Lima, V.L. & de Queiroz, T.A. (2020). Grids for cutting and packing problems: a study in the 2D knapsack problem. 4OR-Q J Oper Res, 18, 293-339. Available at: https://doi.org/10.1007/s10288-019-00419-9
7. Dube, E., Kanavathy, L. R. (2006). OPTIMIZING THREE-DIMENSIONAL BIN PACKING THROUGH SIMULATION. In: Proceedings of the Sixth International Conference, September 11-13, 2006, Gaborone, Botswana. ISBN hardcopy: 0-88986-618-x/cd: 0-88986-620-1.
8. EnzoRuiz (n.d.) GitHub. Retrieved March 6, 2023, from https://github.com/enzoruiz/3dbinpacking
9. Garey, M. R., & Johnson, D. S. (1979). Computers and intractability: A guide to the theory of NP-completeness. San Francisco: W.H. Freeman.

10. Haessler, R.W. (2001). Cutting stock problems. In: Gass, S.I., Harris, C.M. (eds) Encyclopedia of Operations Research and Management Science. Springer, New York, NY. Available at: https://doi.org/10.1007/1-4020-0611-X_203
11. HU, T.C., LANDA, L., SHING, MT. The Unbounded Knapsack Problem. In: Cook, W., Lovász, L., Vygen, J. (eds) Research Trends in Combinatorial Optimization. Springer, Berlin, Heidelberg, 2009. Available at: https://doi.org/10.1007/978-3-540-76796-1_10
12. JamesBremner/knapsack. (n.d.). GitHub. Retrieved March 6, 2023, from https://github.com/JamesBremner/knapsack
13. JOHNSON, David S. Near-optimal bin packing algorithms. Thesis (Ph. D.) - Massachusetts Institute of Technology, Dept. of Mathematics, 1973. Massachusetts Institute of Technology, 1973. Available at: http://hdl.handle.net/1721.1/57819.
14. Kartak, V. M., Ripatti, A. V. The minimum raster set problem and its application to the d-dimensional orthogonal packing problem. European Journal of Operational Research. 2018; 271(1): 33-39. ISSN 0377-2217. Available at: https://doi.org/10.1016/j.ejor.2018.04.046.
15. KELLERER, H., PFERSCHY, U. and PISINGER, D. Knapsack problems. Berlin: Springer, 2004. p. 2. ISBN 978-3-540-40286-2.
16. LAU, H.T. Zero-One Knapsack Problem. In: Combinatorial Heuristic Algorithms with FORTRAN. Lecture Notes in Economics and Mathematical Systems, vol 280. Springer, Berlin, Heidelberg, 1986.Available at: https://doi.org/10.1007/978-3-642-61649-5_3
17. MacLeod, B., Moll, R., Girkar, M., & Hanifi, N. (1993). An algorithm for the 2D guillotine cutting stock problem. European Journal of Operational Research, 68(3), 400-412. ISSN 0377-2217.
18. MARTELLO, S. and TOTH, P. Knapsack Problems: Algorithms and Computer Implementations. Revised edition. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1990. 13 p. ISBN 9780471924203, 0471924202.
19. PISINGER, D. A minimal algorithm for the Bounded Knapsack Problem. In: Balas, E., Clausen, J. (eds) Integer Programming and Combinatorial Optimization. IPCO 1995. Lecture Notes in Computer Science, vol 920. Springer, Berlin, Heidelberg, 1995. Available at: https://doi.org/10.1007/3-540-59408-6_44

20. PISINGER, D. and TOTH, P. Knapsack Problems. In: Du, DZ., Pardalos, P.M. (eds) Handbook of Combinatorial Optimization. Springer, Boston, MA, 1998. https://doi.org/10.1007/978-1-4613-0303-9_5
21. abez, T. A. de, Miyazawa, F. K., Wakabayashi, Y., Xavier, E. C. (2012). Algorithms for 3D guillotine cutting problems: Unbounded knapsack, cutting stock and strip packing. Computers & Operations Research, 39, 200-212. ISSN 0305-0548
22. Scheithauer, G., & Terno, J. (1996). The G4-heuristic for the pallet loading problem. Journal of the Operational Research Society, 47, 511-522.
23. Sgall, J. (2014). Online Bin Packing: Old Algorithms and New Results. In: Beckmann, A., Csuhaj-Varjú, E., Meer, K. (eds) Language, Life, Limits. CiE 2014. Lecture Notes in Computer Science, vol 8493. Springer, Cham. Available at: https://doi.org/10.1007/978-3-319-08019-2_38.
24. Skackauskas, J., Kalganova, T. (2022). Dynamic Multidimensional Knapsack Problem benchmark datasets. Systems and Soft Computing, 4, 200041. ISSN 2772-9419. Available at: https://doi.org/10.1016/j.sasc.2022.200041.
25. WANG, FS. and CHEN, LH. Heuristic Optimization. In: Dubitzky, W., Wolkenhauer, O., Cho, KH., Yokota, H. (eds) Encyclopedia of Systems Biology. Springer, New York, NY, 2013.Available at: https://doi.org/10.1007/978-1-4419-9863-7_411