Master Thesis Applied Mathematics

An efficient bin-packing algorithm for packing groceries in a fulfillment center

Margot van Aken

Delft University of Technology

Faculty Electrical Engineering, Mathematics and Computer Science

March 8, 2019

TU Delft

An efficient bin-packing algorithm for packing groceries in a fulfillment center

Thesis for the
Delft Institute of Applied Mathematics
in order to obtain

the degree of

MASTER OF SCIENCE
in
APPLIED MATHEMATICS

by

Margot van Aken

Student number:	4161785	
Date:	March 8, 2019	
Committee:	dr. ir. F. M. de Oliveira Filho,	TU Delft, supervisor
	prof. dr. ir. K. I. Aardal,	TU Delft
	assoc. prof. dr. ir. M. B. van Gijzen,	TU Delft
	ir. F. B. Gorte,	Picnic, supervisor

TU Delft

Abstract

Picnic is an online supermarket that delivers groceries, packed in totes. We have tried to create a bin-packing algorithm that assigns items from a customer-order to totes such that the amount of totes is minimized. Analyzing the bin-packing algorithm that was used before this thesis had been written, taught us that [redacted]% of the customer-orders was packed non optimal. In this thesis four algorithms are applied to Picnic data. The order in which the algorithms assign items to a tote has major consequences for the solutions. Eight different ways to order the items are combined with each algorithm, resulting in 32 different tote-calculations. Out of those 32 tote-calculations, the Best Fit Algorithm with items ordered in decreasing normalized values generates the best results. Remarkably, ordering items randomly also gives good solutions. This brought us to introducing a new method, where each customer-order is calculated at most eight times, each time shuffling the items before rerunning the algorithm and remembering the better solution. This heuristic is optimal for 99.97% of the customer-orders. An analysis on the financial impact showed us that implementing the new heuristic can save Picnic more than [redacted] in 2019.

Preface

Before you lies the thesis "Picnic's tote-calculation", that came into being in collaboration with the online supermarket Picnic. It has been written to complete the master Applied Mathematics at TU Delft. This research aims to minimize the amount of bins used to ship Picnic's customer-orders. I constructed the algorithm that assigns the groceries to the bins. I loved writing my thesis about a problem that was on one hand mathematically challenging, but on the other hand practical and easy to explain. Also, the direct impact that I could achieve motivated me to think broader than the original subject.

I could not have completed this thesis without the help of many others. I would like to thank my daily supervisor, Fernando Mario de Oliveira Filho, for his feedback and critical questions. Also I would like to thank my committee, Karen Aardal and Martin van Gijzen, for their time and feedback. Special thanks to my Picnic supervisor Frank Gorte from whom I have learned so much.

Furthermore I would like to thank my colleagues at Picnic for their help and for all the fun I had in this period. Especially Jenneke Evers with whom I discussed my progress during a biweekly catch up and Julia van den Belt who reviewed my text many times. Besides the TU and Picnic I would like to thank my friends and family. Furthermore, I would like to thank Hylke Waalewijn, whose broad interest I admire. Even though my field of expertise differs so much from his, he was able to give me accurate advise. My parents deserve a particular note of thanks: their wise counsel, sympathetic ear and cute but sometimes wrong mathematical advice kept me happy and motivated.

I hope you will enjoy reading as much as I have enjoyed writing.

Margot van Aken
Amsterdam, February 18, 2019

Contents

1 Introduction

Online shopping has become a fast-growing market in the Netherlands. “In 2018, the average online market share of non-daily consumer goods was 15.1%. In the fashion industry it was even 28%. Remarkably, the market share of online grocery shopping stays far behind: in 2018 it was only 2%” [1]. However, the revenue in the supermarket sector in the same year was €39 billion [2]. This is where the founders of Picnic saw an opportunity. Their vision is that without expensive physical stores and with an optimized logistics system, high quality groceries can be delivered for free and at a low price. In just a few years Picnic has become a serious challenger in the online supermarket sector in the Netherlands.

This chapter will start with further introducing the online supermarket Picnic, focused on the logistical aspects. Then the subject of this thesis will be given, followed by a short introduction of the bin-packing problem. Eventually a brief outline of all chapters in this thesis is given.

1.1. Picnic’s logistics

Without physical shops, Picnic has a different logistics system than traditional supermarkets, see Figure 1.1. Customers place orders via an application on their mobile phone. The producers deliver goods at the Fulfillment Centers (FCs). At these FCs the customer-orders are packed into two types of totes (crates). Ambient products are packed in plastic red totes, and chilled and frozen products in insulated black totes. These totes are delivered by Picnic employees in electrical vehicles to the customers’ homes. Since the grocery market is a low-margin industry, it requires a relentless focus on cost. The key to become profitable is having a highly efficient supply chain. Picnic’s supply chain has three main aspects:

1. **Accurate forecasting.** Because the exact customer-orders for the next day are known, purchasing products can be done very precisely. This results in low waste.

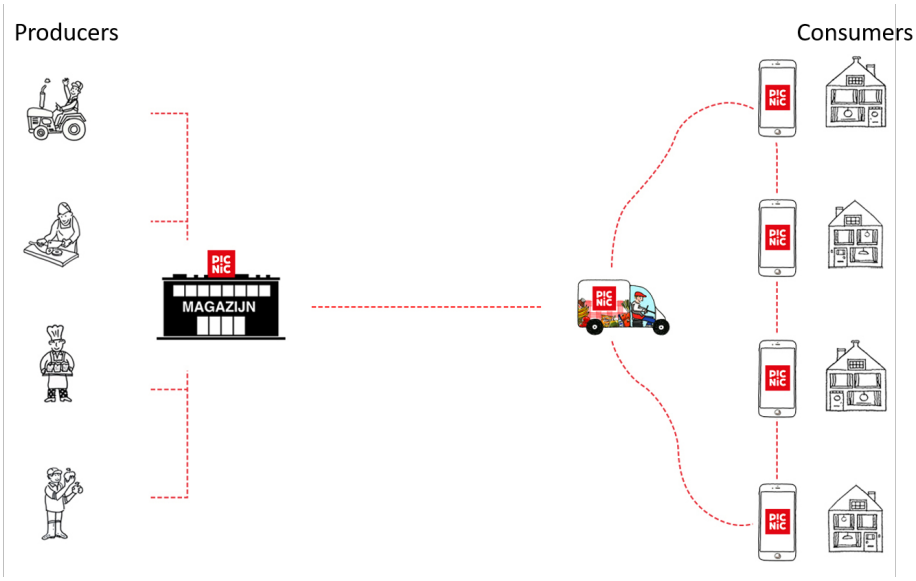


Figure 1.1: Supply chain of Picnic - an online supermarket without physical shops.

2. **Efficient operations in the Fulfillment Center.** The flow of goods in the FCs is illustrated in Figure 1.2. The goods are delivered by producers at the FC, where they are put in logically ordered aisles with shelves. The customer-orders are packed into appropriate totes by Picnic employees, and then put in dispatch frames. In the back of Figure 1.3 the aisles with shelves where the customer-orders are picked are shown. In the front we see the dispatch frames with full totes that are ready to be delivered. These dispatch frames are designed to perfectly fit into the electrical vehicles, see Figure 1.4.

Figure 1.2: Flow of goods through the Fulfillment Center.

3. **Efficient distribution system.** When the groceries are packed, they are ready to be delivered at the customers' homes. Picnic's electrical vehicles are used to deliver the groceries at a fixed time each day of the week. An example of a schedule for a customer could be Monday 15:30 - 16:30, Tuesday 21:00 - 22:00, Wednesday 18:15 - 19:15, and so on. The time slots give the opportunity to cluster deliveries and thereby minimize the distribution costs. The Picnic vehicle routing method was aimed to be optimized in a thesis written by a colleague mathematics graduate intern in 2016, Joris van Tatenhove [4].

Figure 1.3: Picnic's Fulfillment Center.

Figure 1.4: Electrical Picnic vehicle.

1.2. Thesis subject

In this thesis, it is tried to optimize a part of the second aspect of the supply chain: packing customer-orders in the FCs. The problem is as follows: a customer orders a set of products. Each product has a weight and a volume. The totes in the FCs can be filled up to a certain volume and a weight. The goal is to minimize the number of totes used to ship the customer-order. This minimization-problem is known as

the *bin-packing problem* and further introduced in the next section. Optimizing the packing-process can be divided into three subproblems:

1. **Fill-rate optimization.** Currently a fill-rate constraint is implemented in the bin-packing algorithm. A tote that is filled up to 85% of its actual volume capacity is considered full. This fill-rate is meant to prevent overfull totes that can lead to delay in the logistics process. On the other hand, delivering totes with items that could have been packed in less totes is obviously a waste of money. An optimal balance needs to be found between too full and too empty totes. An analysis on the financial impact of increasing the fill-rate is described in the last chapter, but it should still be analyzed which fill-rate is optimal. The question is if the artificial fill-rate of 85% is optimal. The theoretical output should be compared to the practical output of the bin-packing algorithm.

2. **Data accuracy.** Accurate information about the volume and weight of each product is necessary to calculate how many totes are needed to ship a customer-order. Until recently, the data Picnic used was based on supplier data. Due to a lack of incentive to provide accurate data, the data was often not reliable. Improving data accuracy will reduce the margin of error and therefore improve the correctness of the bin-packing algorithm. The fill-rate explained above, can be increased by increasing the data accuracy. This subproblem was tackled in the form of a short internship project. There is a chapter about this project in Appendix A.2.

3. **Bin-packing algorithm.** A bin-packing algorithm runs to calculate which item will be put in which tote. The amount of totes used to ship a customer-order should be minimized, because of costs that linearly scale with the amount of totes used. Before this thesis, there was no data available on the performance of the current algorithm. The main focus of this thesis is Picnic's bin-packing algorithm. The current algorithm is analyzed and a heuristic is created that consistently finds good solutions.

1.3. Bin-packing problem

The bin-packing problem is a well-studied problem in mathematics and computer science with many applications. In each application a set of items is given and the question is what the minimum number of bins is that we need, to fit in all the items. Each application can have additional constraints. You can think of packing goods in trucks in a stable way: heavy products on the bottom, light products on top. Another example is the commercial break between a TV show. A collection of ads of different lengths is packed into the (various) sizes of the break-length. This thesis is about a classical variant of the 2D bin-packing problem, provided in Definition 1.

Definition 1 (2D bin-packing problem)**.** *Given a volume capacity $V \geq 0$, a weight capacity $W \geq 0$, and n items with volumes $v_1, ..., v_n$ and weights $w_1, ..., w_n$. The bin-packing problem is to find the minimum M with an M-partition $B_1 \cup \cdots \cup B_M$ of the set $\{1, ..., n\}$ such that $\sum_{i \in B_k} v_i \leq V$ and $\sum_{i \in B_k} w_i \leq W$ for all $k = 1, ..., M$.*

An optimal solution can be found by simply applying *brute-force*: try all possibilities and pick the best option. The amount of possibilities of placing n items into bins is equal to the amount of all partitions of n items. A short illustration follows. Assume we have a set of items: $\{a, b, c\}$. Place a in bin 1: $\{(a)\}$. There are 2 possibilities to place b: place it in the same bin or in a new bin. For c there are 5 options: $\{(a, b, c)\}, \{(a, b)(c)\}, \{(a, c)(b)\}, \{(a), (b, c)\}$ and $\{(a), (b), (c)\}$. The number of all partitions is called the *Bell number*, shown in Table 1.1.

						n	**Bell number**
1						1	1
1	2					2	2
2	3	5				3	5
5	7	10	15			4	15
15	20	27	37	52		5	52
52	67	87	114	151	203	6	203

Table 1.1: The Bell number up to $n = 6$.

The triangle on the left can be obtained by first putting a 1 at the top. Then the first (leftmost) number of a new row is the rightmost number of the previous row. Other numbers in the triangle are the sum of the number to its left and the number above the number to its left. With only 15 items there are over a billion partitions. In this thesis, we are looking for a smart way to pack customer-orders in as few totes as possible in acceptable little time.

1.4. Report outline

In this section, the subjects of each chapter are shortly described, together with the relevance to the thesis. In Figure 1.5 the thesis structure is given.

Chapter 2 is devoted to complexity theory. Basic concepts of complexity theory are given, leading to Cook's theorem that states that the SATISFIABILITY problem is *NP*-complete. Finally it is proven that the bin-packing problem in the form of a decision problem is *NP*-complete. This proof gives us the clear goal of finding an approximation algorithm in this thesis that computes a solution with an acceptable small error within reasonable time.

Chapter 3 is about a bin-packing problem solving method, called *Bin Completion*. The idea is that a search tree is made with clustered items as nodes. These nodes can be pruned under certain circumstances. The theory in this chapter will give us an example of how to find an optimal solution to a bin-packing problem.

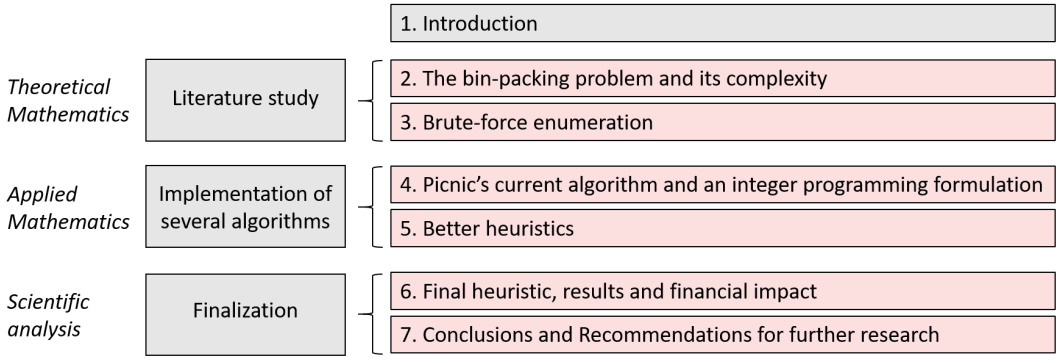


Figure 1.5: Thesis structure.

Furthermore it will give us intuitive insights in why ordering items before executing an approximation bin-packing algorithm can change the output, what we will need later on in the thesis.

The fourth chapter explains the current tote-calculation. First the picking process in the FC is further explained, then the current algorithm is mathematically described, underpinned by an example. Of course we are interested in knowing how good the solutions of the current algorithm are. The solver Gurobi is used to find optimal solutions for a set of [redacted] customer-orders. In order to be able to write a Python script that solves bin-packing instances using Gurobi, an Integer Programming formulation is required. This IP formulation is given in the Subsection 4.3.1. The Gurobi results serve two purposes. Firstly, the current tote-calculation is evaluated and analyzed. Secondly, the results form our main goal of this thesis: write an algorithm that finds solutions that are (almost) as good as the Gurobi solutions.

In the fifth chapter, four different algorithms are given, explained and compared. Since the order of the items influences the output of an algorithm, eight different item-orders are given. These item-orders and algorithms are combined, resulting in 32 different tote-calculations.

In chapter 6, the results are discussed of applying the tote-calculations on Picnic data. Remarkably, the solutions of the algorithms when ordering the items randomly, is a very accurate approximation of the optimal solution. This insight caused the introduction of a new method where the same algorithm runs several times. Before rerunning the algorithm, the order of the items is shuffled. This method turns out to give an optimal output for [redacted] of the customer-orders. The chapter ends with an analysis on the financial impact of implementing this method. More than [redacted] euros can be saved in 2019 by implementing the new heuristic.

The last chapter gives conclusions and recommendations. Various interesting topics for further research occurred during writing the thesis. The recommendations vary from adjusting the artificial volume capacity of a tote to redistributing items between the totes of a customer-order.

2

The bin-packing problem and its complexity

This chapter is devoted to complexity theory. Some basic theory of the complexity classes is given in the first section. In the second section the foundation of all *NP*-complete proofs is given: Cook's Theorem. With this theorem we will prove the complexity of bin-packing.

It is important to know that bin-packing is *NP*-complete, because then we know that there does not exist an algorithm that solves it in polynomial time, unless $P = NP$. For this reason we aim to write an algorithm that finds solutions with an acceptable small error within reasonable time.

In Appendix A.1 an extended version of this chapter is written. First more basic complexity theory is given there, then it is explained how a Turing machine works. We need this knowledge for the whole proof of Cook's Theorem, which is also given in the appendix. In this chapter, only information is given that is necessary for the proof that bin-packing is *NP*-complete.

2.1. Problem complexity

Before introducing computational complexity, it should be clear what a decision problem is. A *decision problem* is a problem that, given an input, asks a yes-no question. Some examples:

1. Input: $x \in \mathbb{Z}$. Question: "Is x a prime number?"
2. Input: a 9×9 matrix, partly filled in with integers 1,...,9. Question: "Is there a Sudoku solution for the given matrix?"
3. Input: a set of n items with weights $w_1, ..., w_n$ and m bins with weight capacity C. Question: "Can we pack items $1, ..., n$ into m bins?"

If the answer is yes, the instance is called a *yes-instance* and a solution is called a *yes-output*. Note that the last example is a one-dimensional bin-packing problem,

written as a decision problem. There are two questions that indicate the complexity of a decision problem:

1. Can we *solve* the decision problem in polynomial time?
2. Given a yes-output, can we *certify* it in polynomial time?

If a decision problem can be *solved* in polynomial time, the decision problem is in the complexity class *P*. If a yes-output of a decision problem can be *certified* in polynomial time, the decision problem is in the complexity class *NP*. Obviously, $P \subseteq NP$.

Definition 2 (Polynomial reduction)**.** *Given two problems X and Y. We say that X **polynomially reduces** to Y if a polynomial algorithm exists such that the input of the algorithm is a yes-instance of X if and only if the output is a yes-instance of Y.*

Later on in this chapter we will prove that the bin-packing problem, formulated as the decision problem in the third example, is *NP*-complete.

Definition 3 (NP-complete)**.** *A decision problem $D \in NP$ is called **NP-complete** if all other problems in NP polynomially reduce to D.*

It might now be clear that, to prove that bin-packing is *NP*-complete, we need to show that bin-packing is in *NP* and that all other problems in *NP* can be polynomially reduced to bin-packing. It is generally easy to see if a problem is in *NP*. For the second part of the proof we need Cook's theorem, stated in the next section.

2.2. Complexity proof

The foundation of all *NP*-complete proofs lays in a theorem that states that the SATISFIABILITY-problem is *NP*-complete [10]. In this section, the SATISFIABILITY-problem will be explained. Then another decision problem, the *3-Dimensional Matching*, will be described. Through this problem we will eventually be able to prove the complexity of bin-packing.

Satisfiability

The question that needs to be answered for the SATISFIABILITY-problem is, given a collection of clauses over a set of literals, is the formula satisfiable, i.e., is there a truth assignment such that the Boolean function corresponding to the family of clauses is true? Let us take an example that explains the SATISFIABILITY-problem. Assume we have $X = \{x_1, x_2, x_3\}$ and let $\overline{x_i}$ be the negation of x_i for $i = 1,2,3$. Then the set of literals is $L = \{x_1, x_2, x_3, \overline{x_1}, \overline{x_2}, \overline{x_3}\}$. If we now take a collection of clauses $\{\{x_1, x_3\}, \{\overline{x_1}, \overline{x_2}\}, \{\overline{x_1}, x_2, x_3\}\}$, then this corresponds to the Boolean function $(x_1 \wedge x_3) \vee (\overline{x_1} \wedge \overline{x_2}) \vee (\overline{x_1} \wedge x_2 \wedge x_3)$. This collection of clauses is satisfiable, because for example the assignment $(x_1, x_2, x_3) = (true, false, true)$ satisfies the problem. An easy example of a no-instance would be the collection of clauses $\{\{x_1\}, \{\overline{x_1}\}\}$. Since $\overline{x_1}$ is the negation of x_1, no assignment can be found such that the Boolean function $(x_1) \vee (\overline{x_1})$ is true.

Theorem 1 (Cook, 1971)**.** *Satisfiability is NP-complete.*

The complete proof of Cook's theorem is given in Appendix A.1.3. By Cook's theorem, it follows from Definition 3 that all problems in *NP* polynomially reduce to SATISFIABILITY. Now consider an arbitrary problem $D \in NP$. If we can find a polynomial reduction from SATISFIABILITY to D, then D must be *NP*-complete. This is the method we will use in this chapter to prove that bin-packing is *NP*-complete. First another decision problem will be described.

3DM.
The question that needs to be answered for the 3DM problem is, given 3 disjoint sets and a set of edges that match one element from each set to each other, can we find a subset of these edges such that all elements are matched exactly once? In Figure 2.1 two examples of the 3DM instance are illustrated. A more formal definition of the 3DM problem follows.

Definition 4 (3-Dimensional Matching)**.** *Given 3 disjoint sets X, Y, Z of equal cardinality and a set of triples $T \subseteq X \times Y \times Z$. Is there a subset $S \subseteq T$ such that $|S| = |X|$ and for each pair $(x, y, z), (x', y', z') \in S$ one has $x \neq x', y \neq y'$ and $z \neq z'$?*

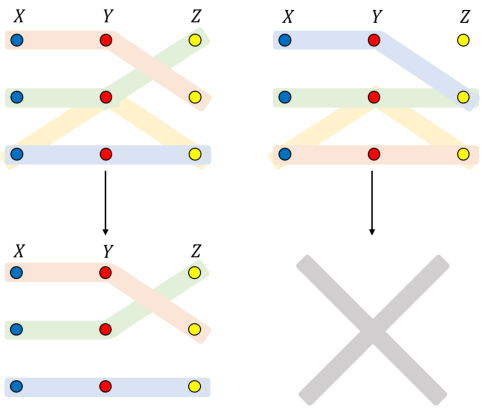
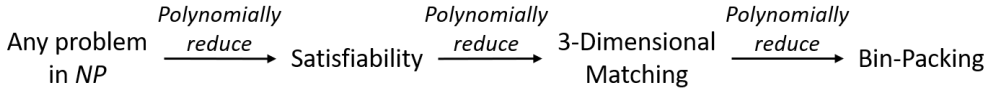


Figure 2.1: Two examples of a 3DM instance.

Theorem 2 (Karp, 1972)**.** *3DM is NP-complete.*

In the proof of this theorem, SATISFIABILITY is reduced to 3DM, see Appendix A.1.3. We now know that SATISFIABILITY can be reduced to 3-Dimensional Matching. Next, we will reduce 3-Dimensional Matching to Bin-packing. Then bin-packing must be *NP*-complete. The complete proof structure is thus the following:



2.2.1. Bin-packing

We have arrived at the most important theorem of this chapter for this thesis. The complexity of bin-packing will finally be showed. The proof of the following theorem is based on a proof written by Papadimitriou in 1982 [13].

Theorem 3. *Given an instance I of a bin-packing problem. It is NP-complete to decide whether I can be solved with B bins.*

Proof. Clearly the problem is in *NP*, because we can easily verify in polynomial time that a given partition is feasible. It leaves us to show that all other problems in *NP* polynomially reduce to this decision problem. We shall reduce 3-Dimensional matching to it.

Given the sets of nodes $X = \{x_1, ..., x_n\}$; $Y = \{y_1, ..., y_n\}$ and $Z = \{z_1, ..., z_n\}$ and the set of triples $T = \{t_1, ..., t_m\} \subseteq X \times Y \times Z$. We are asked whether there is a set of n triples in T, such that each node in X, Y and Z is contained in exactly one of the n triples.

We will construct an instance of bin-packing that has $N = 4m$ items. Each triple in T corresponds to an item, denoted simply t_j. Furthermore, each occurrence of a node in X, Y, and Z to a triple corresponds to an item. Let $u_i \in X \cup Y \cup Z$. Then u_i will be denoted by $u_i[1], u_i[2], ..., u_i[N(u_i)]$, where $N(u_i)$ is the number of occurrences of u_i in the triples. The sizes of the items are shown below:

$$\text{triple } (x_i, y_j, z_k) \in T \text{ has weight } 10M^4 + 8 - iM - jM^2 - kM^3$$

$$x_i[q] \text{ has weight } \begin{cases} 10M^4 + iM + 1 \text{ if } q = 1 \\ 11M^4 + iM + 1 \text{ if } q > 1 \end{cases}$$

$$y_j[q] \text{ has weight } \begin{cases} 10M^4 + jM^2 + 2 \text{ if } q = 1 \\ 11M^4 + jM^2 + 2 \text{ if } q > 1 \end{cases}$$

$$z_k[q] \text{ has weight } \begin{cases} 10M^4 + kM^3 + 4 \text{ if } q = 1 \\ 8M^4 + kM^3 + 4 \text{ if } q > 1 \end{cases}$$

Here, M is a very large number, say $100n$. Note that there is a difference in size between (arbitrarily) the first occurrence and the other occurrences of the nodes in X, Y and Z. Define the bin capacity $C := 40M^4 + 15$. This capacity makes it possible to fit exactly one triple and one node of all three sets X, Y and Z as long as the nodes are either all three or none of the three a first occurrence. There are m bins, as many as triples.

Assume all items fit into m bins. Note that the sum of all items is mC. Hence, each bin must be full. Also, note that the weight of each item is strictly between $\frac{1}{5}$ and $\frac{1}{3}$. Hence, each bin must contain four items. We have $C \mod M = 15$. Given are the numbers 1, 2, 4, and 8. Even if we allow repetition, there is only one way to get the number 15 by choosing four numbers: each number must be chosen once. Furthermore, the sum modulo M^2 must also be 15, so $(i' - i) \cdot M + 15 = 15 \mod M^2$, thus $i = i'$. Similarly, taking the sum modulo M^3 and M^4, get $j = j'$ and $k = k'$. Each bin thus contains a triple $t = (x_i, y_j, z_k)$, together with $x_i \in X$, $y_j \in Y$

and $z_k \in Z$. Furthermore, since $40M^4$ must be reached, either all three occurrences are first occurrences or none of them are. Hence, there are n bins that contain only first occurrences, the n triples in these bins form a 3-Dimensional matching.

Conversely, assume a 3-Dimensional matching exists. Making sure that the triples in the matching get first occurrences of all three members, we can fit all items into the m bins by matching each triple with occurrences of its members. ■

Picnic's bin-packing problem is two-dimensional (each item has a weight and a volume). Corollary 2 states that the decision problem version of the two-dimensional bin-packing problem is also *NP*-complete. It is therefore our goal to write an algorithm in this thesis that finds solutions with an acceptable small error instead of finding a solution, assuming $P \neq NP$.

Corollary 1. *Given an instance of a two-dimensional I bin-packing problem. It is NP-complete to decide whether I can be solved with B bins.*

Proof. Clearly it is in *NP* to decide whether I can be solved with B bins. It leaves us to show that it is *NP*-complete. Consider a bin-packing instance I_1 that is one-dimensional; each item has a volume and each bin has a volume capacity V. Adding weight 0 to each item and adding the weight capacity W to each bin gives us an instance of a two-dimensional bin-packing problem. Obviously this is a yes-instance if and only if I_1 is a yes-instance. We see that one-dimensional bin-packing is a special case of two-dimensional bin-packing. We may conclude that it is *NP*-complete to decide whether a two-dimensional bin-packing problem instance can be solved with B bins. ■

3

Brute-force enumeration

An instance of a bin-packing problem can be solved by simply applying brute-force. As explained in the introduction, the calculation time explodes if the amount of items increases. To drastically decrease calculation time, Fukunaga and Korf [17] developed a method that combines a bin-oriented branch-and-bound strategy and dominance criteria [18] and called it *Bin Completion*. The idea is that a search tree is made with clustered items as nodes. These nodes can be pruned under certain circumstances. This chapter is devoted to Fukunaga and Korf's theory. It is assumed that the standard principles of the branch-and-bound method are known [20].

The first section will explain how to make a basic search tree that finds solutions to a bin-packing problem. In the second section, a proposition is given that states certain criteria of when to prune the search tree. In the third section, a method is constructed where only bin assignments are found that do not needs to be pruned according to the theory in this chapter. The fourth section then combines the previous sections to the final method: bin completion.

3.1. Bin-oriented search tree

This section will first explain how to make a basic search tree that leads to every feasible solution of a bin-packing problem. In the first subsection, the nodes in the search tree represent items that need to be packed. In the second subsection, certain items are clustered to reduce calculation time.

3.1.1. Item-oriented search tree

Let us start with an example that will be the common thread in this chapter. Assume we have a set I of 8 items with weights 10, 10, 8, 8, 7, 7, 6, and 3 and a bin capacity of 20. Without loss of generality, we can order the items in non-increasing weight. As explained in Chapter 1, the total amount of possible bin assignments, including infeasible bin assignments, is the Bell number. For $n = 8$ this is 4140. An explanation of a search tree with all 4140 bin assignments follows. Place the first

item in a bin: (10). For the second item there are two possibilities: 1. Place it in the same bin: (10,10). 2. Place it in a new bin: (10)(10). For the third item there are five possibilities: (10,10,8); (10,10)(8) and (10,8)(10); (10)(10,8); (10)(10)(8). Continuing this way will give us the 4140 options.

Definition 5 (Bin assignment)**.** *Given a set of items $I = \{1, ..., n\}$ with weights $\{w_1, ..., w_n\}$ and a bin with capacity C. A bin assignment A is a subset $A \subseteq I$. A* ***bin assignment*** *is feasible if $\sum_{i \in A} w_i \leq C$ and infeasible otherwise.*

Going back to the example, the bin assignment (10,10,8) is infeasible. In consequence, the following bin assignments are also infeasible: (10,10,8,8) and (10,10,8)(8). To find only feasible solutions, prune under infeasible bin assignments. In Figure 3.1 this has been done for the first four items. This method will lead to 45 feasible bin assignments. In the figure each node represents how an item can be put in a bin, given that the items up to the current item are put in bins according to what is shown in the previous node. Each level i in the search tree shows partitions of the first i items.

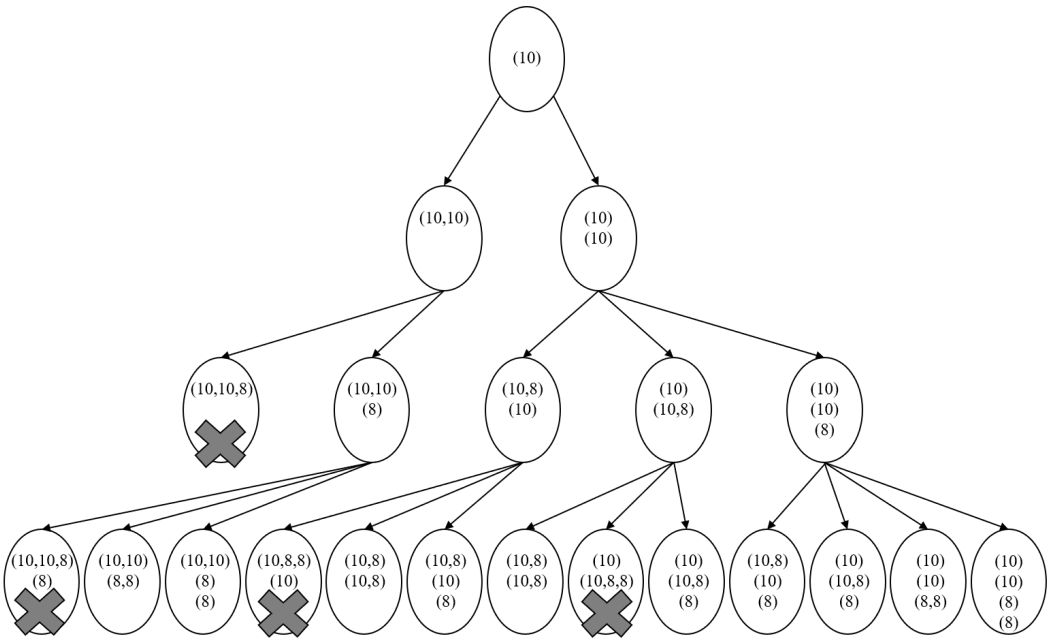


Figure 3.1: Part of an item-oriented search tree for items with weights 10, 10, 8, 8, 7, 7, 6, and 3 and bin capacity 20.

3.1.2. Bin-oriented search tree

Note that this item-oriented method needs much calculation time for larger sets of items. This is because every item is assigned to a bin one at a time. What if the items can be clustered? In a bin-oriented branch-and-bound algorithm, items are clustered in *maximal* subsets.

Definition 6 (Maximal)**.** *Given a set of items I with weights $\{w_1, ..., w_n\}$ and a bin capacity C. A feasible set of items $M \subseteq I$ is called* maximal *if there does not exist an item $i \in I \backslash M$ such that $M \cup \{i\}$ is feasible.*

In other words, a feasible set is **maximal** if adding *any* item to the set makes it infeasible. Maximal sets of the example set I including the first item 10, are $(10,10)$; $(10,8)$; $(10,7,3)$ and $(10,6,3)$. A search tree with maximal bin assignments is called a *bin-oriented* search tree. A bin-oriented search tree of the example is shown in Figure 3.2. Where in Figure 3.1 the nodes represent partitions, in Figure 3.2 each node represents a maximal bin assignment of the sub problem excluding the items in the bin shown in the previous node. The maximal bin assignments include the first item of the (sub)problem. Example: the first maximal bin assignment $(10,10)$ leaves the remaining subproblem 8, 8, 7, 7, 6, and 3. All maximal bin assignments including the first item then are $(8,8,3)$, $(8,7,3)$ and $(8,6,3)$. Still 20 feasible solutions occur. It

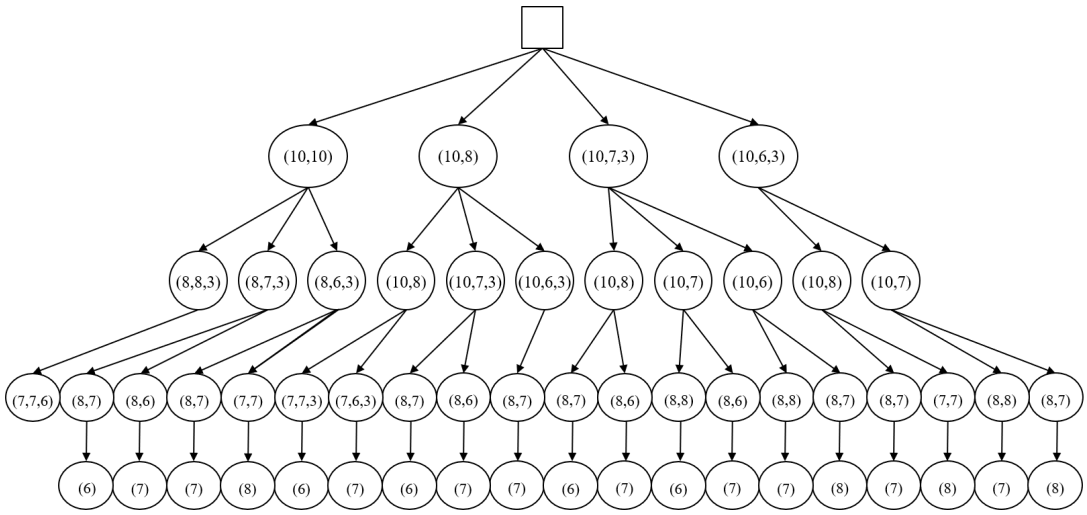


Figure 3.2: Search tree with maximal bin assignments as nodes including the first item of the (sub)problem, for items with weights (10,10,8,8,7,7,6,3) and bin capacity 20.

might be clear that this has to do with exploring what follows from every maximal bin assignment. If we can show that certain maximal bin assignments *always* induce solutions with more bins than other maximal bin assignments, we can prune under those bin assignments.

3.2. Dominance criteria

This section is devoted to the definition of *dominance* and to a proposition with dominance criteria and their proofs. These dominance criteria will lead to an efficient way to prune under certain bin assignments.

Definition 7 (Dominant)**.** *Given two bin assignments A and B, we say that A **dominates** B, if the optimal solution after assigning A consists of at most as many bins as the optimal solution after bin assignment B.*

We need to find criteria that show the dominance of certain bin assignments. The first obvious dominance criterion is maximality. If a bin assignment is not maximal, it is by definition a subset of another feasible bin assignment and is therefore dominated.

An intuitive example that shows other dominance criteria follows. Again consider the same example set I with weights 10, 10, 8, 8, 7, 7, 6 and 3, and bin capacity 20. Consider the maximal bin assignments including the first item $A_1 = (10,10)$, $A_2 = (10,8)$, $A_3 = (10,7,3)$ and $A_4 = (10,6,3)$. It is always better if there is an option to divide a weight in the remaining subproblem. Therefore A_1 dominates A_3. Furthermore, if we compare A_1 with A_2, we see that all elements of A_2 can be mapped one-to-one to the elements of A_1, and the elements of A_2 are all at most as big as the elements of A_1. The remaining subproblem after assigning A_2 contains as many items as after assigning A_1, but all remaining items are at least as big as the remaining items after assigning A_1. Therefore A_1 dominates A_2. Combining this reasoning gives that A_1 also dominates A_4: $\{10\} \mapsto 10$ and $\{6,3\} \mapsto 10$. These criteria are formulated by Christofides, Mingozzi, and Toth [19] and by Martello and Toth [18]:

Proposition 1 (Dominance Criteria)**.** *Given two feasible bin assignments A and B.*

1. *A dominates B if $B \subseteq A$.*
2. (Christofides, Mingozzi and Toth) *A dominates B if there exists a one-to-one mapping from B to A such that each item $b \in B$ is mapped to an element of $a \in A$ with $w(b) \leq w(a)$.*
3. (Martello and Toth) *A dominates B if B can be partitioned into n subsets $B_1, ..., B_n$ such that each subset B_i is mapped one-to-one to an item $a_i \in A$ with $w(B_i) \leq w(a_i)$.*

Proof. Consider a set of items I and bin assignments A and B.

1. Assume bin assignment B is a subset of bin assignment A and let $I' = \{x_1, ..., x_n\}$ be the set of elements such that $I' \cup B = A$. Consider an optimal solution including bin assignment B, using M bins. Moving all elements of I' to the bin containing B, leads to another feasible solution containing bin assignment A. We have shown that M is the upper bound of an optimal solution containing bin assignment A. Hence, A dominates B.
2. The proof of criterion 2 follows from the proof of criterion 3.
3. Assume B can be partitioned into n subsets $B_1, ..., B_n$ such that each subset B_i is mapped one-to-one to an item $a_i \in A$ with $w(B_i) \leq w(a_i)$. Given B, let s_B be an optimal solution, with $|s_B|$ bins. Now consider the bins with the items of A. Swap the items of A with the corresponding partitions of B. After each swap, all bins will remain feasible, since A is feasible and $w(B_i) \leq w(a_i)$. Hence, there is a solution containing bin assignment A with $|s_B|$ bins, so $|s_B|$ is the upper bound of an optimal solution containing bin assignment A.

■

Note that any bin assignment that would be pruned by the second criterion, would also be pruned by the third criterion. This does not apply the other way around.

Definition 8 (Strongly dominant)**.** *Given two bin assignments A and B, we say that A **strongly dominates** B, if A dominates B conforming the Dominance Criteria.*

3.3. Finding non strongly dominated bin assignments

The problem remains to find non strongly dominated bin assignments. This section will explain how to find them in two steps: first find maximal bin assignments and then find the non strongly dominated bin assignments.

3.3.1. Finding maximal bin assignments

All items must be assigned to a bin eventually. It therefore has no influence on the total amount of bins in a solution when choosing one item and find all maximal bin assignments including that item. See the first part of Proposition 1 for further explanation. Also, the less the remaining capacity after choosing the first item, the fewer possible maximal bin assignments can be found, hence the less computation time needed. Therefore order the items in decreasing weight and take the first item as start item.

Make a binary tree with the first item as root. Each node represents an item. Left branches represent including the item and right branches represent excluding the item. The root therefore only has a left branch. The leafs represent maximal bin assignments. In Figure 3.3 the example is shown. For each node calculate the summed weight. This is the weight of nodes with a left branch plus the weight of the current node. Also determine for each branch which items can potentially be added to the bin without exceeding the bin capacity. If the weight of the remaining items together with summed weight is less than or equal to the bin capacity, make a leaf with all those items. The leafs are maximal bin assignments.

3.3.2. Eliminate dominated bin assignments

Using Proposition 1, we could compare all maximal bin assignments and eliminate the strongly dominated ones. The number of maximal bin assignments is potentially much bigger than the number of items. Furthermore, we can only compare them, if we store them. However, we could also compare a maximal bin assignment to items that are not included. Then we can immediately eliminate strongly dominated bin assignments. This saves calculation time and memory storage. Let W be the sum of the weights of a maximal bin assignment A and let C be the bin capacity. Note that the bin assignment A is strongly dominated if and only if there is an item $x \in I \backslash A$ such that replacing a subset $y \subseteq A$ with x increases W without exceeding C. So

$$W \leq W - w(y) + w(x) \leq C.$$

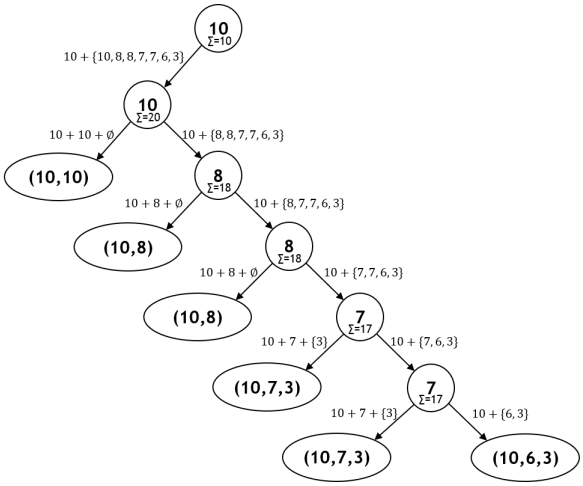


Figure 3.3: Binary search tree aiming to find all maximal bin assignments including the first item.

Using this elimination formula, the binary tree that finds only non strongly dominated bin assignments is drawn in Figure 3.4.

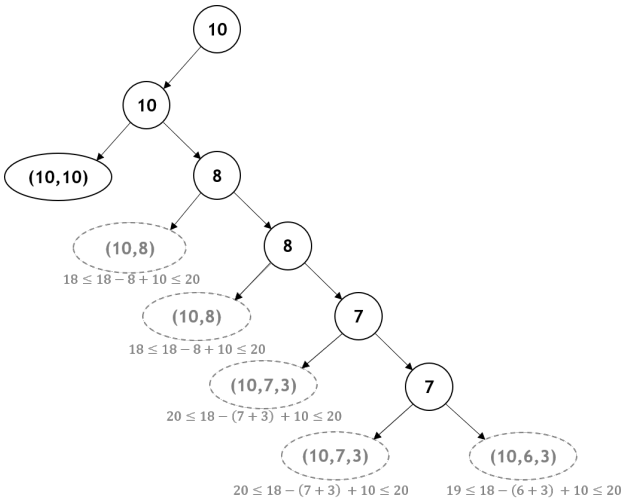


Figure 3.4: Binary search tree with non strongly dominated bin assignments as leaves. The grey dashed leafs are strongly dominated maximal bin assignments.

3.4. Bin completion

Combining the previous three sections gives bin completion: a bin-oriented branch-and-bound method with maximal bin assignments as nodes, where the dominance criteria are used to prune the search tree. All nodes of the branch-and-bound tree are non strongly dominated bin assignments. Other maximal bin assignments are pruned. If several non strongly dominated bin assignments exist, the trivial lower bound should be calculated to see under which node should be searched first.

Definition 9 (Trivial lower bound)**.** *Given a bin-packing problem for a set of items I with weights $\{w_1, ..., w_n\}$ and a bin capacity C. Let $w(I) = \sum_{i=1}^{n} w_i$. The* ***trivial lower bound*** *of the problem is:*

$$LB(I) = \left\lceil \frac{w(I)}{C} \right\rceil .$$

In the example we have been using this chapter, we see that in each step only one non strongly dominated bin assignment exists. Therefore the bin completion tree for this example is simple, see Figure 3.5. The trivial lower bound is only of im-

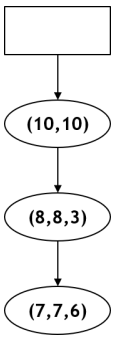


Figure 3.5: Bin completion tree for weights $\{10, 10, 8, 8, 7, 7, 6, 3\}$ and bin capacity 20.

portance if several non strongly dominated bin assignments exist with the first (remaining) item. An example follows. Assume we have a set of 9 items with weights 10, 9, 9, 9, 6, 5, 5, 5, and 2 and a bin capacity of 20. The total weight is 60, so we have a trivial lower bound of $\lceil \frac{60}{20} \rceil = 3$ bins. With the method explained in Subsection 3.3.1 and 3.3.2, we find two non strongly dominated bin assignments including the first item: (10,9) and (10,5,5). Calculating the lower bound again gives us $LB_{(10,9)} : 1 + \lceil \frac{41}{20} \rceil = 4$ and $LB_{(10,5,5)} : 1 + \lceil \frac{40}{20} \rceil = 3$. Because $3 < 4$, we first search under the bin assignment (10,5,5). The next step in the bin completion method is thus to find non strongly dominated bin assignments of the items with weights 9, 9, 9, 6, 5, and 2. As long as in each step the lower bound is at most 4, we do not need to explore the possibilities under bin assignment (10,9). This eventually gives us the tree in Figure 3.6.

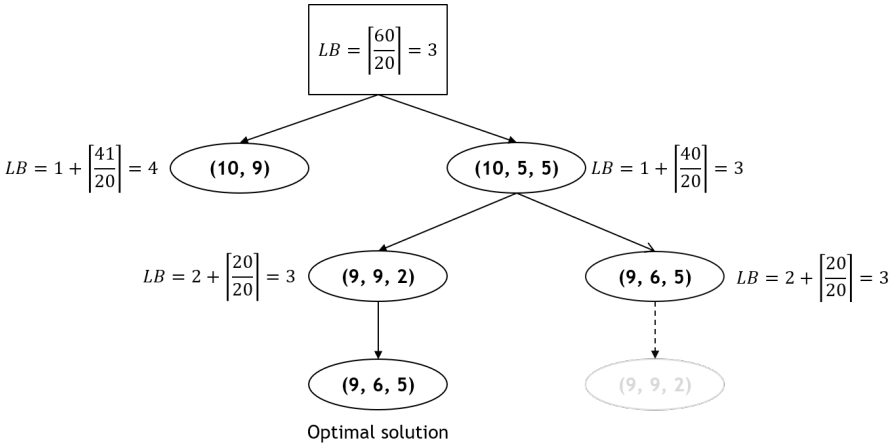


Figure 3.6: Bin completion tree for weights $\{10,9,9,9,6,5,5,5,2\}$ and bin capacity 20.

4

Picnic's current algorithm and an integer programming formulation

In this chapter, the current bin-packing algorithm Picnic uses will be described. To fully understand what the bin-packing algorithm is used for, it is explained in the first section what the picking process in a fulfillment center looks like. In the second section the current algorithm will be explained and an upper bound is given. In the third section the current algorithm will be analyzed and compared to optimal solutions obtained by the solver Gurobi. For this an integer programming formulation is written, given in Subsection 4.3.1. All results in this chapter are based on approximately 21 thousand customer-orders of Picnic; all the customer-orders on a Monday and a Friday in October 2018. Monday and Friday provide most interesting data, because firstly, these days are the busiest. Secondly, the size of an average customer-order on Monday is smaller than an average customer-order on Friday. This difference might give interesting results.

4.1. Picking process

This section will provide insights in how groceries are picked such that they can be delivered at the customers' homes. The explanation of the picking process will be divided into three parts.

Tote

A tote is a crate that is used for delivering the customer-orders. A customer-order can be packed in several totes, this is referred to as a multiple tote customer-order. Each tote contains items of only one customer. In other words, if a customer orders groceries that fill 1.2 totes, there will be 2 totes shipped to the customer. In each tote there are three bags placed. The totes are used to pack and distribute the bags with groceries. The customer only receives the bags. There are two different types of totes. Open, red totes are used for storing ambient items and black totes are

used for storing chilled and frozen items and closed with a lid after the picking process.

Picking path

All goods are put in logically ordered aisles with shelves. The order the items are placed in is determined by four item characteristics: volume, weight, fragility, and contamination. The items' volume and weight get smaller as the path approaches the end. Also, since it became clear to Picnic that clients do not like their groceries to be packed such that chemical items touch their food, employees get the instruction to pack all contaminating items in the middle bag. To make this instruction as easy as possible for the employee, all contaminating items are placed in the first aisle. Furthermore, fragile items are placed at the end of the picking path. Because this path is followed, a light, small and fragile item is never put under a heavy item that could damage it. A simplified illustration of a picking path is shown in Figure 4.1.

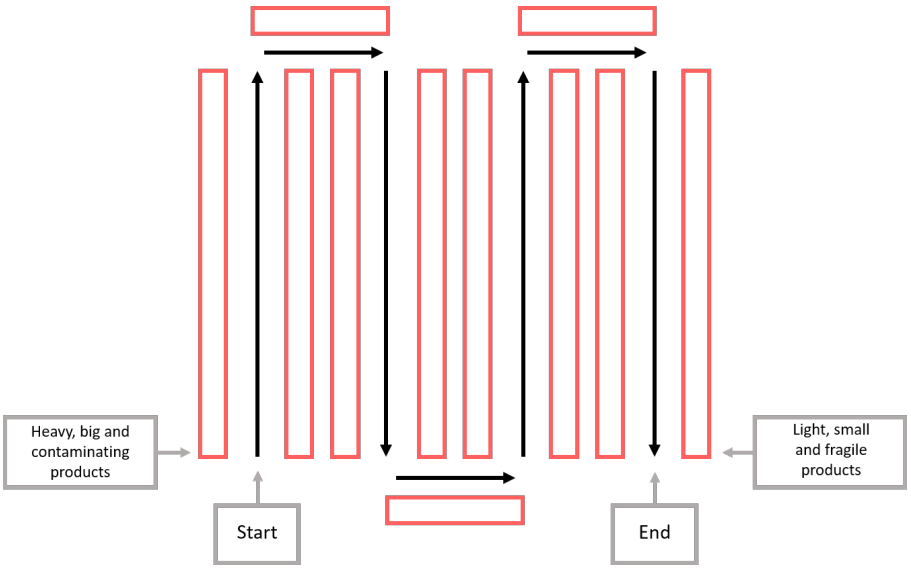


Figure 4.1: Illustration of a picking path.

Order picking

A pick cart holding multiple totes is pulled by an employee through the picking path to pack items into totes, see Figure 4.2. The employee gets instructions from a device on her arm. It tells her which item to pack in which tote. Throughout the process, every time an item is placed in a tote, both the item and the tote must be scanned. If a mistake is made, this is indicated by the device. Therefore, wrong items that actually arrive at a customer are rare. Once the picking process has been completed, the totes are put into specially designed frames, that fit into the vehicles that deliver the groceries to the customers. These frames are called dispatch frames.

Figure 4.2: An employee picking groceries.

Initially, order picking and dispatch frame loading were managed by third-party software. This software lacked flexibility for process improvements on various topics, like allocation of totes to pick carts. In the summer of 2018, Picnic launched its own, in-house designed Warehouse Management System (WMS). This system manages all processes within the fulfillment center. Because of time constraints, the WMS has adopted the former tote allocation strategy for now. The goal of this thesis is to analyze the bin-packing algorithm written in WMS and if possible improve it. In other words, our goal is to optimize the output of the device that gives instructions to the employees during the picking process.

4.2. Current bin-packing algorithm

Now that the picking process is explained, the current bin-packing algorithm can be described. The current algorithm Picnic uses is a fast and simple algorithm, also known as the *Next Fit Algorithm*. First the mathematical formulation of the algorithm is given, followed by an example. Afterwards it is explained how well the algorithm theoretically performs.

Current bin-packing algorithm. Given a set of items $I = 1, ..., n$ with corresponding volumes $v_1, ..., v_n$ and weights $w_1, ..., w_n$, ordered in decreasing weight and given totes y_j, $j \geq 1$ with a volume capacity V and a weight capacity W. This algorithm assigns each item to a tote such that the items will be placed in one tote until either V or W is reached. Then the next item will be assigned to a new tote. Variables v_a and w_a are used for auxiliary storage; they respectively represent the summed volume and weight of the items in the tote to which the last item is assigned.

1. [$Initialize$.] Set $j = 1$, $v_a = v_1$ and $w_a = w_1$. Assign 1 to y_1.

2. [$Assign\ the\ items\ to\ a\ tote$.] For $i = 2, ..., n$, if $v_a + v_i \leq V$ and $w_a + w_i \leq W$, assign i to y_j, set $v_a = v_a + v_i$ and $w_a = w_a + w_i$. Otherwise, set $j = j + 1$,

assign i to y_j and set $v_a = v_i$ and $w_a = w_i$. Repeat step until every item is assigned to a tote.

This algorithm runs in $O(n)$ time. Above we see the mathematical formulation of the algorithm. To analyze and use it, it is written in a Python script.

An example of how the algorithm works follows. For the sake of an intuitive illustration, we assume that ordering in decreasing weight is equal to ordering in picking path location. Given a customer that ordered items $1,...,9$ with corresponding volumes $(v_1,...,v_9) = (65,60,55,50,45,40,40,20,15)$ and weights $(w_1,...,w_9) = (5,5,4,3,3,3,2,2,1)$. The way the items are placed in the picking path can be seen in Figure 4.3. Apply the algorithm, assuming a tote can be filled up to volume $V = 200$

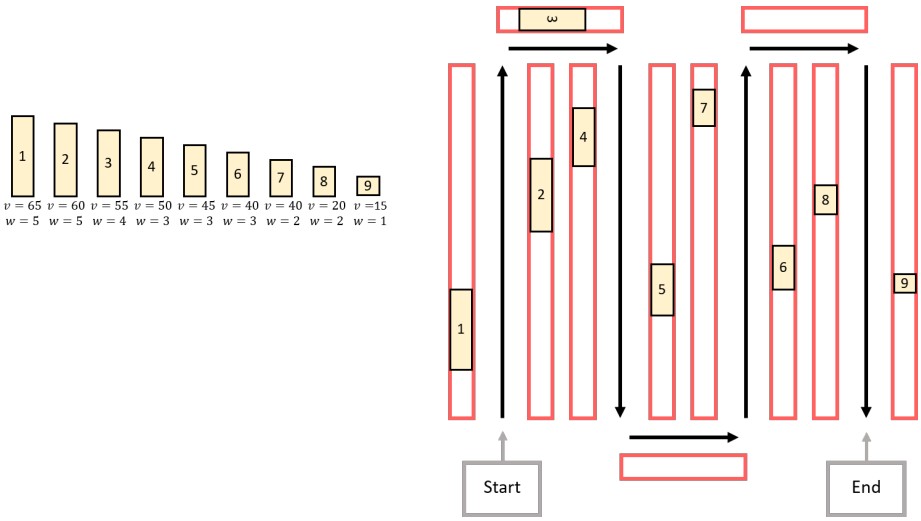


Figure 4.3: Example of one customer-order in a fulfillment center

and weight $W = 18$. Also assume there is an unlimited supply of totes y_j, $j \geq 1$ and consider auxiliary variables v_a and w_a.

1. Set $j = 1$, $v_a = v_1 = 65$ and $w_a = w_1 = 5$. Assign 1 to y_1.
2. $v_a + v_2 = 125 \leq V$ and $w_a + w_2 = 10 \leq W$, so assign 2 to y_1. Set $v_a = 125$ and $w_a = 10$.
3. $v_a + v_3 = 180 \leq V$ and $w_a + w_3 = 14 \leq W$, so assign 3 to y_1. Set $v_a = 180$ and $w_a = 14$.
4. $v_a + v_4 = 230 \not\leq V$, so assign 4 to y_2. Set $v_a = v_4 = 50$ and $w_a = w_4 = 4$.
5. $v_a + v_5 = 96 \leq V$ and $w_a + w_5 = 7 \leq W$, so assign 5 to y_2. Set $v_a = 95$ and $w_a = 7$.
6. $v_a + v_6 = 135 \leq V$ and $w_a + w_6 = 10 \leq W$, so assign 6 to y_2. Set $v_a = 135$ and $w_a = 10$.

7. $v_a + v_7 = 175 \leq V$ and $w_a + w_7 = 12 \leq W$, so assign 7 to y_2. Set $v_a = 175$ and $w_a = 12$.

8. $v_a + v_8 = 195 \leq V$ and $w_a + w_8 = 14 \leq W$, so assign 8 to y_2. Set $v_a = 195$ and $w_a = 14$.

9. $v_a + v_9 = 210 \not\leq V$, so assign 9 to y_3.

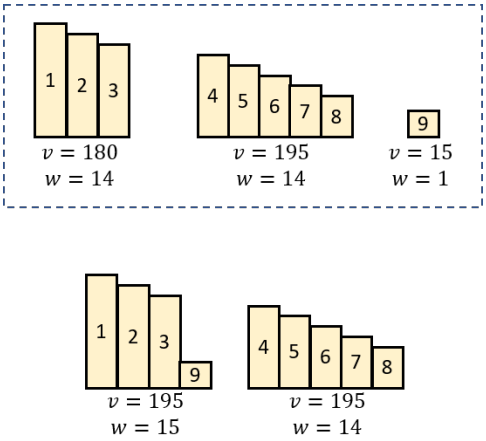


Figure 4.4: Output of the algorithm in dashed lines: three totes. A better solution is also given, using two totes.

The solution that the algorithm gives is shown in Figure 4.4. Clearly, the algorithm does not find a solution with the minimum amount of totes, since item 9 could have been assigned to tote y_1.

Following is an intuitive analysis of how well the algorithm performs. It is divided into the amount of totes the algorithm uses to assign all items of a customer-order to.

- **One tote.** If the current algorithm assigns all items to one tote it is optimal, since one tote is the absolute minimum.

- **Two totes.** If the algorithm assigns all items to two totes, it means that either the maximum volume or the maximum weight of the fist tote has been reached. Any bin-packing algorithm that minimizes the number of totes would need a second tote. Therefore we can conclude that solutions where the items are assigned to a total of two totes are also optimal solutions.

- **Three or more totes.** Assume the algorithm assigns the items to three or more totes. Items will only be assigned to the current or to a new tote. If an item is assigned to a new tote, it means that the space left in the current tote is less than the space needed for *that* item. This does not mean that *no* item fits in that tote. This is what we saw in the example in the previous section. We conclude that solutions where the items are assigned to three or more totes, are potentially non-optimal solutions.

Because all potential optimization is in customer-orders that are currently packed in three or more totes, we should look at the distribution of amount of totes per customer-order, shown in Figure 4.5. We see that the percentage of customer-orders that are currently packed in one or two totes is 94%. The percentage of customer-orders that could potentially have a reduction in totes is 6%.

Amount of totes per customer-order:	**Percentage of total amount of customer-orders:**
1	[redacted]%
2	[redacted]%
3	[redacted]%
4	[redacted]%
>4	[redacted]%

Figure 4.5: Distribution of the amount of totes used per customer-order.

Knowing the distribution given in Figure 4.5, it is not that odd that Picnic did not improve its tote calculation yet. The current algorithm provides the absolute minimum amount of totes for at least [redacted]% of the customer-orders in linear time.

There is a noticeable difference between the two days that are analyzed. The customer-orders on Mondays are smaller than on Fridays. This can be seen in the data that is used for this research. The percentage of customer-orders that is packed in one or two totes on the Monday that was analyzed was [redacted]% and on Friday [redacted]%.

4.3. Determine optimality using Gurobi

To verify the performance of the current algorithm, we need to know the absolute minimum number of totes per customer-order. For this thesis the solver Gurobi is used in the programming language Python. Gurobi is a commercial optimization solver that solves among others mixed-integer programming (MIP) problems. These MIP problems are generally solved using Branch-and-Bound. For the implementation of the solver, the 2D bin-packing problem has to be written as an integer programming problem. This section starts therefore with a subsection containing an integer programming formulation. Finally, the results obtained by Gurobi are given.

4.3.1. Integer Programming formulation

The input of Picnic's bin-packing problem is a customer-order; a set I of items with $|I| = n$. A volume function $v : I \to \mathbb{R}$ assigns a volume to each item and a weight function $w : I \to \mathbb{R}$ assigns a weight to each item. A supply of n totes is given. Each tote can be filled up to weight W and volume V. The goal is to minimize the amount of totes that are used to pack all items.

This bin-packing problem should be translated into an integer programming formulation in a systematic way. First the variables need to be defined. These variables are used to formulate the constraints corresponding to the feasible solutions of the problem and to formulate the objective function.

Definition of the variables.

$$x_{ij} = \begin{cases} 1 & \text{if item } i \text{ is packed in tote } j \\ 0 & \text{otherwise} \end{cases}, \quad y_j = \begin{cases} 1 & \text{if tote } j \text{ is used} \\ 0 & \text{otherwise.} \end{cases}$$

Definition of the constraints.
Each item is placed in a tote:

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \text{for } i = 1, ..., n.$$

The volume in each tote is less than or equal to a maximum volume V:

$$\sum_{i=1}^{n} x_{ij} v_i \leq V \quad \text{for } j = 1, ..., n.$$

The weight in each tote is less than or equal to a maximum weight W:

$$\sum_{i=1}^{n} x_{ij} w_i \leq W \quad \text{for } j = 1, ..., n.$$

An item is only assigned to a tote if that tote is in use:

$$x_{ij} \leq y_j \quad \text{for } i = 1, ..., n \text{ and } j = 1, ..., n.$$

Definition of the objective function.
The amount of totes used to pack all items is minimized:

$$\min \sum_{j=1}^{n} y_j.$$

Now the integer programming formulation is complete for the bin-packing problem. Extra constraints to the interests of Picnic can be applied later. The resulting

IP is:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{j=1}^{n} y_j \\
\text{subject to} \quad & \sum_{j=1}^{n} x_{ij} = 1 \quad \text{for } i = 1, ..., n \\
& \sum_{i=1}^{n} x_{ij} v_u \leq V \quad \text{for } j = 1, ..., n \\
& \sum_{i=1}^{n} x_{ij} w_i \leq W \quad \text{for } j = 1, ..., n \\
& x_{ij} \leq y_j \quad \text{for } i = 1, ..., n \text{ and } j = 1, ..., n \\
& x_{ij} \in \{0, 1\}, \quad y_j \in \{0, 1\}
\end{aligned}
$$

4.3.2. Gurobi output

The integer program is written in a Python script to solve a set of 21 thousand customer-orders using the Gurobi optimizer. The results can be seen in Table 4.1. The second column tells us how many totes are actually needed if the current algorithm packs the items in the amount of totes that is written in the first column. Indeed we see that if the current algorithm assigns all items to one or two totes, it

Current algorithm:	**Gurobi:**
1	1
2	2
3	2.56
4	3.34
5	4.27
6	5.09
...	...

Table 4.1: Gurobi results.

is optimal. The more totes used, the worse the algorithm's solution is. To see what this outcome means for the two analyzed days at Picnic, the Table 4.2 is attached.

We see that there is a potential tote reduction of [redacted] totes for the two days that are analyzed. In total there are [redacted] customer-orders that could have been packed in less totes; only eight customer-orders have a potential tote reduction of more than one tote. The results in percentages:

- [redacted]% of the totes do not need to be used.
- [redacted]% of the customer-orders can be packed in less totes.

As explained at the end of Section 4.2, customer-orders on Fridays are bigger than on Mondays. The logical consequence is that there are more non-optimal solutions made by the current algorithm on Friday than on Monday. On the analyzed

Amount of totes:	**Amount of customer-orders:**	**Total amount of totes used:**	**Gurobi:**
1	[redacted]	[redacted]	[redacted]
2	[redacted]	[redacted]	[redacted]
3	[redacted]	[redacted]	[redacted]
4	[redacted]	[redacted]	[redacted]
5	[redacted]	[redacted]	[redacted]
6	[redacted]	[redacted]	[redacted]
>6	[redacted]	[redacted]	[redacted]
Total:	**20861**	**29570**	**28907**

Table 4.2: The amount of totes Picnic used and the amount of totes they actually needed.

Monday there is a potential tote reduction of [redacted]% of the customer-orders, against [redacted]% on the analyzed Friday. Chapter 5 is devoted to finding a way to achieve this reduction.

5

Better heuristics

In this chapter, four algorithms and item-orders are given and analyzed. The first section is devoted to these algorithms: *Next Fit*, *First Fit*, *Best Fit* and *Worst Fit*. They are described, mathematically formulated and explained through an example. Also, they are compared based on item distribution, calculation time and performance. The subject of the second section is item-ordering. It is explained why ordering items affects the output of an algorithm. All different item-orders that we use in this thesis are given.

5.1. Four bin-packing heuristics

In this section, four algorithms will be given and explained. To prevent repetition of the same introduction in each following subsection, it is given here:

- **Mathematical formulation.** For the mathematical formulation of the algorithm, assume a set of items $I = 1, ..., n$ is given with corresponding volumes $v_1, ..., v_n$ and weights $w_1, ..., w_n$ and an unlimited amount of totes y_j, $j \geq 1$ with volume capacity V and weight capacity W. Variables $v(y_j)$ and $w(y_j)$ represent the volume and weight of tote y_j.
- **Example set.** One simplistic example is given, applied to each algorithm. To provide a simple, but satisfying example, weights are omitted. The example set consists of 10 items with volumes $(v_2, ..., v_{10}) = (8,3,8,3,8,2,2,7,7,2)$. The totes $(y_1, y_2, ...)$ have volume capacity 10. The example set is given in Figure 5.1.

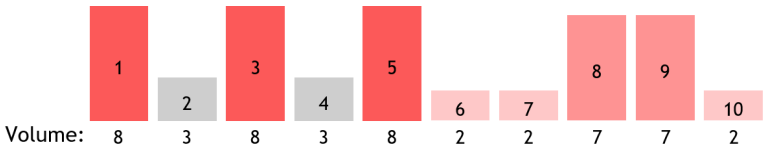


Figure 5.1: Example set.

5.1.1. Next Fit Algorithm

The current algorithm is *Next Fit*. Since the mathematical formulation is described in Section 4.2, here only the example from Figure 5.1 is given.

***Next Fit* example.** Consider the example set. *Next Fit* will do the following:

1. Set $j = 1$, $v_a = v_1 = 8$. Assign item 1 to y_1.
2. $v_a + v_1 = 11 > 10$, so assign item 2 to y_2. Set $v_a = v_2 = 3$.
3. $v_a + v_3 = 11 > 10$, so assign item 3 to y_3. Set $v_a = v_3 = 8$.
4. $v_a + v_4 = 11 > 10$, so assign item 4 to y_4. Set $v_a = v_4 = 3$.
5. $v_a + v_5 = 11 > 10$, so assign item 5 to y_5. Set $v_a = v_5 = 8$.
6. $v_a + v_6 = 10$, so assign item 6 to y_5. Set $v_a = v_a + v_3 = 10$.
7. $v_a + v_7 = 12 > 10$, so assign item 7 to y_6. Set $v_a = v_7 = 2$.
8. $v_a + v_8 = 9 \leq 10$, so assign item 8 to y_6. Set $v_a = v_a + v_8 = 9$.
9. $v_a + v_9 = 16 > 10$, so assign item 9 to y_7. Set $v_a = v_9 = 7$.
10. $v_a + v_{10} = 9 \leq 10$, so assign item 10 to y_7. Done.

We see that *Next Fit* packs the items of the example into 7 totes. The example is illustrated in Figure 5.2.

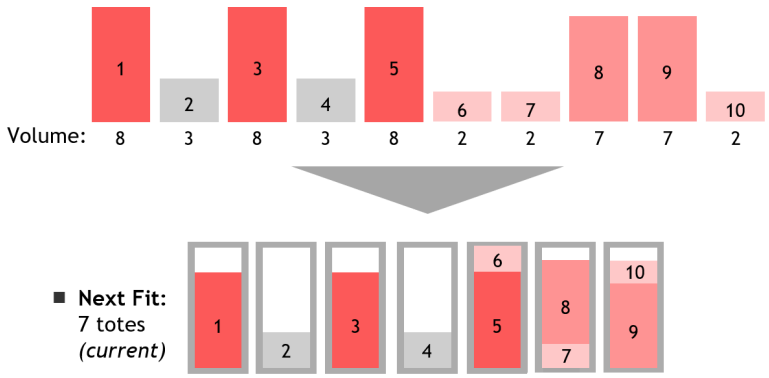


Figure 5.2: *Next Fit* example.

5.1.2. First Fit Algorithm

An obvious improvement of *Next Fit* would be to try to assign the item in previous totes as well instead of only to the current tote. *First Fit* does exactly that. It assigns each item to the first feasible tote. If no tote is found, the next item will be assigned to a new tote.

First Fit Algorithm Given the assumptions for the mathematical formulation written in the introduction of this section.

1. [$Initialize$.] Set $v(y_1) = v_1$ and $w(y_1) = w_1$. Assign 1 to y_1. Set $m = 1$.

   For $i = 2, ..., n$ execute step 2 and 3:

2. [$Assign\ item\ to\ the\ first\ feasible\ tote$.] For $j = 1, ..., m$ if $v(y_j) + v_i \leq V$ and $w(y_j) + w_i \leq W$, assign i to y_j, set $v(y_j) = v(y_j) + v_i$ and $w(y_j) = w(y_j) + w_i$, **break**.

3. [$If\ no\ tote\ is\ found,\ assign\ item\ to\ a\ new\ tote$.] Otherwise, set $m = m + 1$, assign i to y_m and set $v(y_m) = v_i$ and $w(y_m) = w_i$.

First Fit Example Consider the given example. *First Fit* will do the following:

1. Set $j = 1$, $v(y_1) = v_1 = 8$. Assign item 1 to y_1. Set $m = 1$.
2. $v(y_1) + v_2 = 11 > 10$, so assign item 2 to y_2. Set $v(y_2) = 3$ and $m = 2$.
3. $v(y_1) + v_3 = 16 > 10$, $v(y_2) + v_3 = 11 > 10$, so assign item 3 to y_3. Set $v(y_3) = 8$ and $m = 3$.
4. $v(y_1) + v_4 = 11 > 10$, $v(y_2) + v_4 = 6 \leq 10$, so assign item 4 to y_2. Set $v(y_2) = 6$.
5. $v(y_1) + v_5 = 16 > 10$, $v(y_2) + v_5 = 14 > 10$, $v(y_3) + v_5 = 16 > 10$, so assign item 5 to y_4. Set $v(y_4) = 8$ and $m = 4$.
6. $v(y_1) + v_6 = 10$, so assign item 6 to y_1. Set $v(y_1) = 10$.
7. $v(y_1) + v_7 = 12 > 10$, $v(y_2) + v_7 = 8 \leq 10$, so assign item 7 to y_2. Set $v(y_2) = 8$.
8. $v(y_1) + v_8 = 15 > 10$, $v(y_2) + v_8 = 15 > 10$, $v(y_3) + v_8 = 15 > 10$, $v(y_4) + v_8 = 15 > 10$, so assign item 8 to y_5. Set $v(y_5) = 7$ and $m = 5$.
9. $v(y_1) + v_9 = 15 > 10$, $v(y_2) + v_9 = 15 > 10$, $v(y_3) + v_9 = 15 > 10$, $v(y_4) + v_9 = 15 > 10$, $v(y_5) + v_9 = 14 > 10$, so assign item 9 to y_6. Set $v(y_6) = 7$ and $m = 6$.
10. $v(y_1) + v_{10} = 12 > 10$, $v(y_2) + v_{10} = 10$, so assign item 10 to y_2. Done.

We see that *First Fit* packs the items of the example into 6 totes. The example is illustrated in Figure 5.3.

5.1.3. Best Fit Algorithm

The *Best Fit* algorithm assigns each item to the *tightest* tote. The tightest tote is the tote with the least space left. If the item does not fit in any tote, it will be assigned to a new tote.

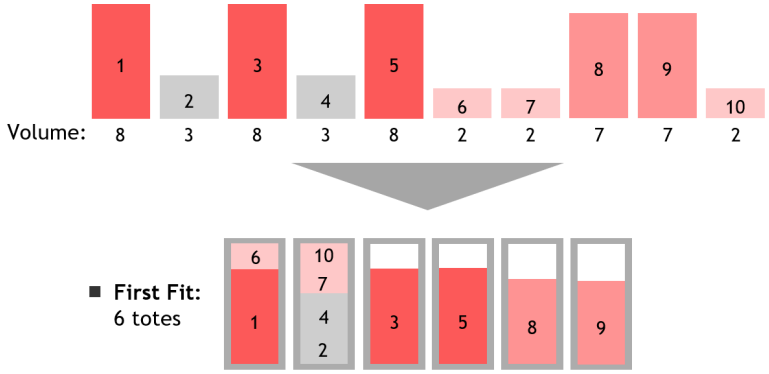


Figure 5.3: *First Fit* example.

Best Fit Algorithm Given the assumptions for the mathematical formulation written in the introduction of this section. Also, variables α, ω_j and Ω are used for auxiliary storage.

1. [*Initialize.*] Set $v(y_1) = v_1$ and $w(y_1) = w_1$. Assign 1 to y_1. Set $m = 1$.

   For $i = 2, ..., n$ execute step 2, 3 and 4:

2. [*Calculate auxiliary volume.*] For $j = 1, ..., m$, if $v(y_j) + v_i \leq V$ and $w(y_j) + w_i \leq W$ set $\omega_j = v(y_j) + v_i$. Otherwise, set $\omega_j = -1$.

3. [*Determine the highest auxiliary volume.*] Set $\Omega = -1$ and $\alpha = m + 1$. For $j = 1, ..., m$, if $\omega_j > \Omega$ set $\Omega = \omega_j$ and $\alpha = j$.

4. [*Assign item to the fullest possible tote.*] If $\Omega = -1$ set $m = m+1$. Assign item i to tote y_α.

Note that this algorithm assigns the items to the tote with most volume. Similarly, the algorithm could assign the items to the tote with most weight, or we could combine weight and volume such that the algorithm assigns the items to the tote with the tightest density. In the results, all possibilities are examined. The same holds for the algorithm described in the next subsection; *Worst Fit*.

Best Fit Example Consider the given example. *Best Fit* will do the following:

1. Set $j = 1$, $v(y_1) = v_1 = 8$. Assign 1 to y_1. Set $m = 1$.

2. $v(y_1) + v_2 = 11 > 10$, so set $\omega_1 = -1$. $\Omega = -1, \alpha = 2$ so set $m = 2$. Assign item 2 to y_2. Set $v(y_2) = 3$.

3. $v(y_1) + v_3 = 16 > 10$, so set $\omega_1 = -1$. $v(y_2) + v_3 = 11 > 10$, so set $\omega_2 = -1$. $\Omega = \max\{-1, -1\} = -1$ so $\alpha = 3$. Set $m = 3$. Assign item 3 to y_3 and set $v(y_3) = 8$.

4. $\Omega = \max\{-1, 6, -1\}$ so $\alpha = 2$. Assign item 4 to y_2. Set $v(y_2) = 6$.

5. $\Omega = \max\{-1,-1,-1\}$ so $\alpha = 4$. Set $m = 4$. Assign item 5 to y_4 and set $v(y_4) = 8$.

6. $\Omega = \max\{10,8,10,10\}$ so $\alpha = 1$. Assign item 6 to y_1. Set $v(y_1) = 10$.

7. $\Omega = \max\{-1,8,10,10\}$ so $\alpha = 3$. Assign item 7 to y_3. Set $v(y_3) = 10$.

8. $\Omega = \max\{-1,-1,-1,-1\}$ so $\alpha = 5$. Set $m = 5$. Assign item 8 to y_5 and set $v(y_5) = 7$.

9. $\Omega = \max\{-1,-1,-1,-1,-1\}$ so $\alpha = 6$. Set $m = 6$. Assign item 9 to y_6 and set $v(y_6) = 7$.

10. $\Omega = \max\{-1,8,-1,10,9\}$ so $\alpha = 4$. Assign item 10 to y_4. Done.

We see that the *Best Fit* algorithm packs the items of the example into 6 totes. The example is illustrated in Figure 5.4.

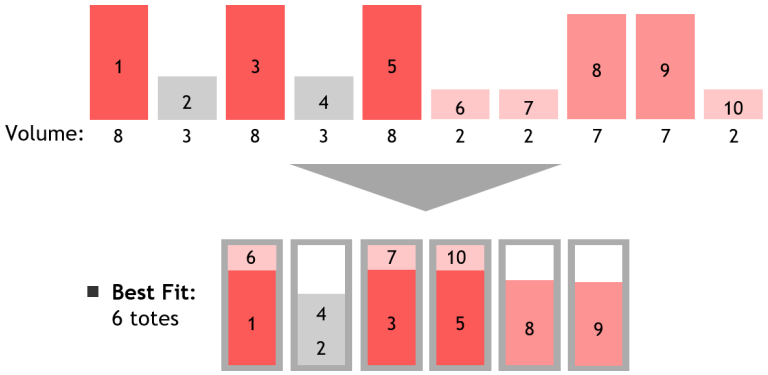


Figure 5.4: *Best Fit* example.

5.1.4. Worst Fit Algorithm

One could say that *Worst Fit* is the inverse of *Best Fit*; it assigns items to the *least tightest* tote. If the item does not fit in any tote, the item will be assigned to a new tote.

Worst Fit Algorithm Again use assumptions given in the introduction of this section. Also, variables α, θ_j and Θ are used for auxiliary storage.

1. [$Initialize$.] Set $v(y_1) = v_1$ and $w(y_1) = w_1$. Assign 1 to y_1. Set $m = 1$.

   For $i = 2,...,n$ execute step 2, 3 and 4:

2. [$Calculate\ auxiliary\ volume$.] For $j = 1,...,m$, if $v(y_j)+v_i \leq V$ and $w(y_j)+w_i \leq W$ set $\theta_j = v(y_j) + v_i$. Otherwise, set $\theta_j = \infty$.

3. [$Determine\ the\ lowest\ auxiliary\ volume$.] Set $\Theta = \infty$ and $\alpha = m+1$. For $j = 1,...,m$, if $\theta_j < \Theta$ set $\Theta = \theta_j$ and $\alpha = j$.

4. [*Assign the item to the most empty tote.*] If $\Theta = \infty$ set $m = m+1$. Assign item i to tote y_α.

Worst Fit Example Consider the given example. *Worst Fit* will do the following:

1. Set $j = 1$, $v(y_1) = v_1 = 8$. Assign item 1 to y_1. Set $m = 1$.
2. $v(y_1) + v_2 = 11 > 10$, so set $\theta_1 = \infty$. $\Theta = \infty, \alpha = 2$ so set $m = 2$. Assign item 2 to y_2. Set $v(y_2) = 3$.
3. $v(y_1) + v_3 = 16 > 10$, so set $\theta_1 = \infty$. $v(y_2) + v_3 = 11 > 10$, so set $\theta_2 = \infty$. $\Theta = \min\{\infty, \infty\} = \infty$ so $\alpha = 3$. Set $m = 3$. Assign item 3 to y_3 and set $v(y_3) = 8$.
4. $\Theta = \min\{\infty, 6, \infty\}$ so $\alpha = 2$. Assign item 4 to y_2. Set $v(y_2) = 6$.
5. $\Theta = \min\{\infty, \infty, \infty\}$ so $\alpha = 4$. Set $m = 4$. Assign item 5 to y_4 and set $v(y_4) = 8$.
6. $\Theta = \min\{10, 8, 10, 10\}$ so $\alpha = 2$. Assign item 6 to y_2. Set $v(y_2) = 8$.
7. $\Theta = \min\{10, 10, 10, 10\}$ so $\alpha = 1$. Assign item 7 to y_1. Set $v(y_1) = 10$.
8. $\Theta = \min\{\infty, \infty, \infty, \infty\}$ so $\alpha = 5$. Set $m = 5$. Assign item 8 to y_5 and set $v(y_5) = 7$.
9. $\Theta = \min\{\infty, \infty, \infty, \infty, \infty\}$ so $\alpha = 6$. Set $m = 6$. Assign item 9 to y_6 and set $v(y_6) = 7$.
10. $\Theta = \min\{\infty, 10, 10, 10, 9, 9\}$ so $\alpha = 5$. Assign item 10 to y_5. Done.

We see that *Worst Fit* packs the items of the example into 6 totes. The example is illustrated in Figure 5.5.

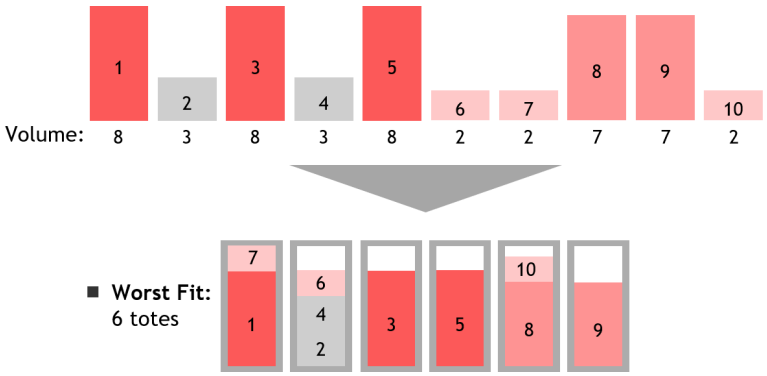


Figure 5.5: *Worst Fit* example.

5.1.5. Comparison of the algorithms

This subsection is devoted to the differences between *Next Fit, First Fit, Best Fit* and *Worst Fit*. Define *Other Fit*:={*First Fit, Best Fit, Worst Fit*}. The most obvious difference between any algorithm in *Other Fit* and *Next Fit* is that while *Next Fit* only takes the current tote into account, the other algorithms also look back to previous totes. Other differences between the algorithms are split into item distribution, calculation time and quality of the solutions.

Item distribution. The difference between the algorithms in *Other Fit* has mostly to do with the distribution of the items over the totes. We can easily see the different distributions when comparing the solutions of the example set used in this section, see Figure 5.6. *First Fit* provides solutions with the first tote the fullest, *Best Fit* provides solutions with some very full totes and some empty totes and *Worst Fit* provides solutions where the items are more evenly distributed over the totes.

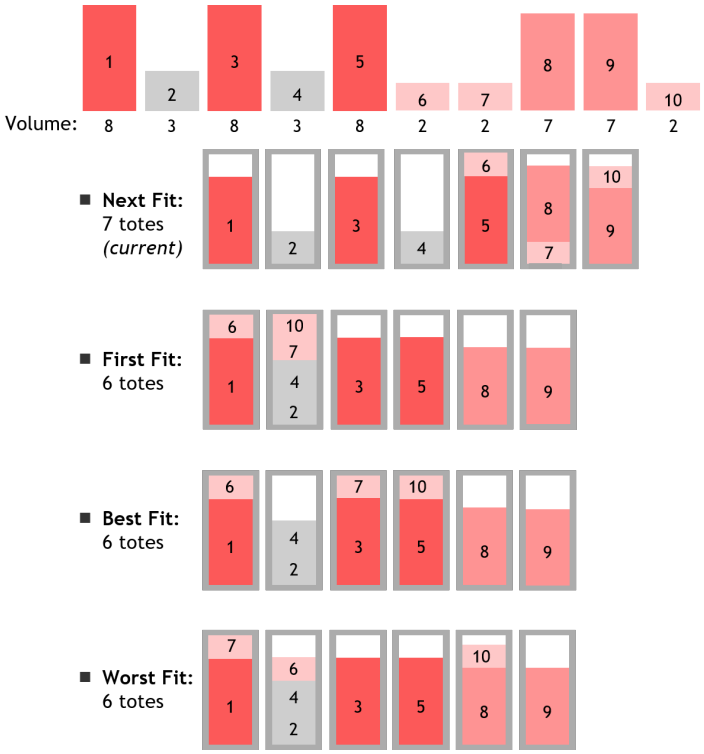


Figure 5.6: Example for *Next Fit, First Fit, Best Fit* and *Worst Fit*.

Calculation time. *Next Fit* runs in $O(n)$ time, while the other algorithms in this chapter run in $O(n^2)$ time. If we store the bins in a self-balancing binary search tree, the algorithm can be implemented in $O(n\log(n))$ time.

Solution quality. It might be clear that *Next Fit* is not as good as any algorithm in *Other Fit*. This is proven in the following lemma.

Lemma 1. *Let I be a bin-packing instance. Let A be an algorithm in Other Fit, then*

$$A(I) \leq NF(I).$$

Proof. Let $1,...,n$ be the items of I. Let $\underline{Y}^a := \{Y_1^a, Y_2^a, ...\}$ be the *Other Fit* tote-assignments and let $\underline{Y}_n := \{Y_1^n, Y_2^n, ...\}$ be the *Next Fit* tote-assignments. If *Next Fit* fills each tote up to its capacity, then $\underline{Y}_a = \underline{Y}_n$ and thus $A(I) = NF(I)$. Assume that $\underline{Y}_a \neq \underline{Y}_n$ and that $i \in I$ is the first item such that $i \in Y_j^a$ and $i \in Y_k^n$, with $j \neq k$. Since *Next Fit* only takes the current tote into account, we know that $j < k$. Consequently, $Y_j^n \subseteq Y_j^a$. By Proposition 1 part 1, Y_j^a dominates Y_j^n, hence $A(I) \leq NF(I)$. ■

Coffman, Garey, Johnson, and Tarjan [22] proved that $FF(I) \leq \frac{17}{10} \cdot OPT(I) + 1$.

Let us take some examples where every time only one algorithm from *Other Fit* is worst . Assume we have totes with volume capacity 20, again weights are omitted. Consider the following volumes:

- $I = (14, 15, 8, 5, 6, 4, 5, 2)$
  First Fit packs the items into **4** totes: (14,5)(15,4)(8,6,5)(2)⟵ worst solution
  Best Fit packs the items into **3** totes: (14,6)(15,5)(8,4,5,2)
  Worst Fit packs the items into **3** totes: (14,4,2)(15,5)(8,5,6)

- $I = (15, 8, 8, 3, 2, 2, 2)$
  First Fit packs the items into **2** totes: (15,3,2)(8,8,2,2)
  Best Fit packs the items into **3** totes: (15,2,2)(8,8,3)(2) ⟵ worst solution
  Worst Fit packs the items into **2** totes: (15,3,2)(8,8,2,2)

- $I = (16, 15, 4, 3, 2)$
  First Fit packs the items into **2** totes: (16,4)(15,3,2)
  Best Fit packs the items into **2** totes: (16,4)(15,3,2)
  Worst Fit packs the items into **3** totes: (16,3)(15,4)(2)⟵ worst solution

Next Fit is not included in the examples above, since by Lemma 1 *Next Fit* is always at most as good as any algorithm in *Other Fit*. The examples serve the purpose of giving the insight that it depends on the set of items which algorithm from *Other Fit* performs best.

5.2. Ordering items

This section is devoted to item-ordering. First an intuitive insight is provided as to why ordering items before executing one of the algorithms could give a different solution. To do so, two simple examples are given of when ordering in decreasing volume gives a better and when it gives a worse solution. Then the examples are explained referring to the dominance criteria from Chapter 3. After these examples, several item-orders are given that are applied to Picnic data in the next section.

- **Example 1.** Consider four items with volumes $(1,2,8,9)$ (no weights), and totes with volume capacity 10. The lower bound is $\lceil \frac{1+2+8+9}{2} \rceil = 2$ totes. All four algorithms will provide the same solution: $(1,2)(8)(9)$. Clearly the first two items can be split over the second and the third tote. Now order the items in decreasing volume: $(9,8,2,1)$. Again *First Fit*, *Best Fit* and *Worst Fit* will provide the same solution, but this time it is an optimal solution: $(9,1)(8,2)$. *Next Fit* will still not give an optimal solution: $(9)(8,2)(1)$.

- **Example 2.** It is not always better to order the items in decreasing volume though. Consider six items with volumes $(2,5,3,4,2,4)$ (no weights), and totes with volume capacity 10. The lower bound is $\lceil \frac{2+5+3+4+3+4}{2} \rceil = 2$ totes. All algorithms will give an optimal solution: $(2,5,3)(4,2,4)$. Now order the items in decreasing volume: $(5,4,4,3,2,2)$. All algorithms will give the following, worse solution: $(5,4)(4,3,2)(2)$.

The theory of the dominance criteria in Chapter 3 might give an intuitive insight in why ordering items in decreasing volume or weight before executing one of the algorithms, often gives a better solution. Consider an example set of volumes $\{4,2,2\}$. Referring to Proposition 1 part 2, it is *always* equally good or better to first assign the 4 and then the two 2's. On the other hand, if we have the example set of volumes $\{4,3,2\}$, Proposition 1 does not tell us anything about it. In other words, it depends on the set of items if ordering in decreasing weight or volume gives better solutions.

The weight of the items does not linearly scale with the volume of the items. Some extreme examples are chips and a bottle of beer. Ordering these items in decreasing weight, means ordering them in increasing volume. We introduce a normalized value that takes both volume and weight into account. Let i be an item with volume v_i and weight w_i. Let V and W be the totes' volume and weight capacity respectively. Define $s_i := \frac{v_i}{V} + \frac{w_i}{W}$. This value makes it possible to order the items in both weight and volume at the same time. This could increase the effect of Proposition 1 part 2 on the solutions.

For this thesis several item-orders are applied to the algorithms described in the previous section. The items are ordered in:

1. Weight - high to low;
2. Weight - low to high;
3. Volume - high to low;
4. Volume - low to high;
5. Normalized - high to low;
6. Normalized - low to high;
7. Shelf location;
8. Random.

As explained in Chapter 4, if a customer-order is assigned by *Next Fit* to one or two totes in total, *all* algorithms from Section 5.1 with *all* item-order written above

will give an optimal solution. Note that for each set of items there exists at least one item-order (not necessarily one that is written above) in which *all* algorithms from Section 5.1 will provide an optimal solution. After all, if *Next Fit* gives an optimal solution, all the other algorithms will do that too.

6

Final heuristic, results and financial impact

In this chapter, the results are described and interpreted in the form of a financial impact. From now on, we call a combination of the algorithms with an item-order from previous chapter a *tote-calculation*. Not only do the tote-calculations described in Chapter 5 need to be compared to each other, we also need to compare their performances to Gurobi (see Chapter 4). The same data set as in Chapter 4 is used in this chapter. The results of the tote-calculations are described in the first section. The second section will then describe the final heuristic. In the third section an analysis is made on the financial impact of implementing the final heuristic.

6.1. Results

In the data set that is used, there is an absolute potential reduction of 663 totes. Define those 663 totes as 100% of the non optimal tote-assignments with respect to the current situation. In Figure 6.1 is an explanatory chart given that explains the charts further on in this chapter. The higher the column, the worse the given calculation. Our goal is to find a calculation that gives zero non optimal tote-assignments. Combining Section 5.1 (four algorithms) and Section 5.2 (eight item-orders), gives us 32 calculations. The results of these calculations are shown in Figure 6.2.

It is to be expected that each time the items are shuffled, the output of the algorithms will be different. For this reason, the calculations with a random item-order were executed 15 times. The differences were small and there were no outliers. Because the biggest difference was less than 0.13% of the total tote amount, taking the average of 15 times reshuffling and rerunning the algorithm gives a good approximation of what the output will be when ordering the items randomly.

In addition, the results are examined of *Best Fit* and *Worst Fit* assigning items to the (least) tightest density, volume and weight. The differences between the results

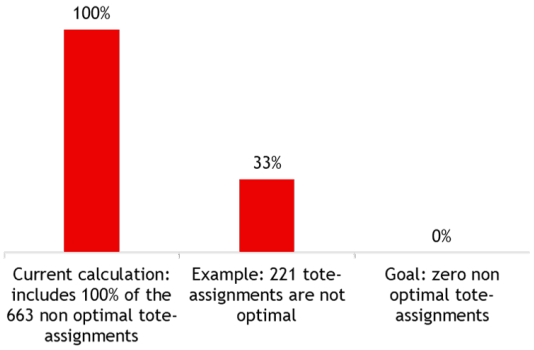


Figure 6.1: Explanatory chart.

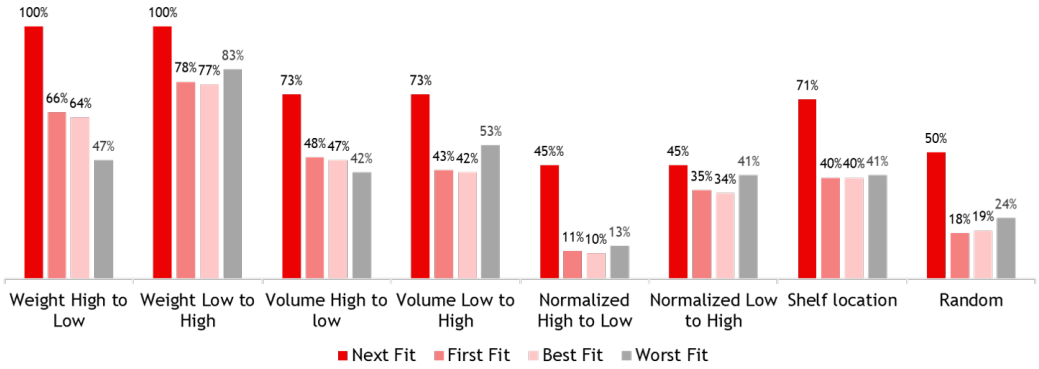


Figure 6.2: Results of 32 different tote-calculations.

were at most $6.9 \cdot 10^{-3}$%, thus negligible. Only assigning items to the (least) tightest volume is therefore shown in the results, see Figure 6.2.

Note that the current calculation (*Next Fit*, items ordered in decreasing weight) is the worst calculation. Furthermore, note that there is no big difference between ordering in *increasing* or *decreasing* volume/weight, in contrast to the normalized value, where ordering in *increasing* or *decreasing* value does differ. As explained in Section 5.2, this has to do with Proposition 1 part 2.

6.2. Final heuristic

Remarkably, shuffling the items gives quite good solutions. Since there always exists an item-order in which all algorithms will output an optimal solution, it might be an idea to combine several item-orders and remember the best solution. Preferably, we do not want to recalculate a customer-order if we already have an optimal solution. We therefore calculate the lower bound (see Definition 9) and only redo the calculation with the items in a different order if the solution does not equal the lower bound.

Final heuristic:

1. Calculate lower bound
2. Shuffle the items
3. Run FIRST FIT
4. Do the following up to 8 times:
   I) If the output of FIRST FIT equals the lower bound:
      i. Stop.
   II) Otherwise:
      i. Shuffle items
      ii. Run FIRST FIT
      iii. Remember output only if it uses less totes

In Chapter 4 we used the solver Gurobi to find the amount of totes needed minimally per customer-order. Comparing the Gurobi-solutions with a simple calculated lower bound shows us that only 0.06% of the customer-orders have an optimal solution that does not equal their lower bound. Furthermore, 97% of the customer-orders were already optimal using *Next Fit*, items ordered in decreasing weight. By Lemma 1 we know that *First Fit* is at least as good as *Next Fit*. Therefore the extra calculation time needed to shuffle and recalculate is low. The results of the method above are shown in Figure 6.3. We see that after reshuffling the

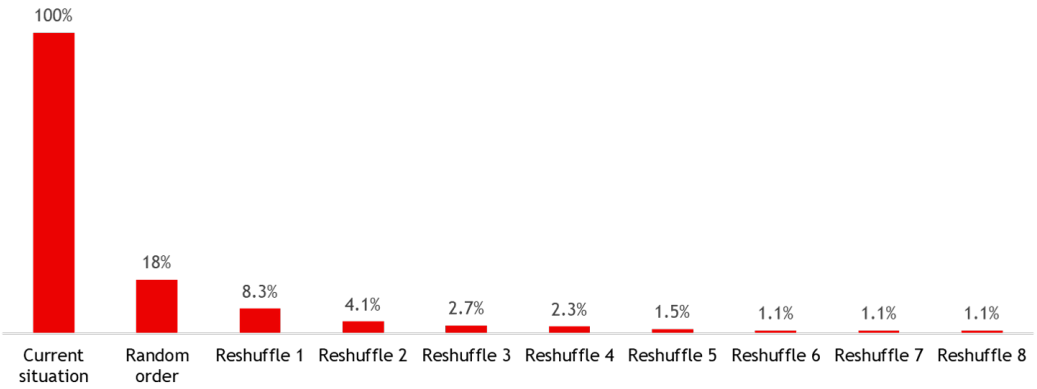


Figure 6.3: Progress of the results when shuffling the items several times.

items 6 times, no further reduction occurs. Out of 20861 customer-orders, there are 7 customer-orders for which Gurobi gives a better solution than the method described in this thesis. In total there is a difference of 7 totes between the Gurobi solutions and the solutions obtained by the new heuristic. We conclude that we have a total tote-reduction of 2.24% and that for 99.97% of the customer-orders the given method is optimal.

6.3. Estimation of the financial impact

The financial impact of implementing the final heuristic written in the previous section, is given in this section. In order to calculate cost savings, we need to know

which costs linearly scale with the amount of totes used per customer-order. Together with a Picnic mathematician from the growth team, an analysis is made on the costs that linearly scale with the amount of totes used. Costs are mostly based on the time needed to fulfill a task, expressed in wage costs (€[redacted]/hour). Other costs are directly linked to the costs of certain products (like bags or vehicles). These costs can be divided into fulfillment and distribution costs. In Figure 6.4 a chart is displayed that shows how these costs are made up. Important for now is that each tote extra costs €[redacted].

Figure 6.4: Costs that linearly scale with the amount of totes used.

Cost savings in 2019. In this subsection, it will be calculated how much money will be saved in 2019 by implementing the new heuristic. For the two days that are analyzed, [redacted] totes could have been saved by the new heuristic. That gives us an average of [redacted] saved totes per day. This is equivalent to a cost-saving of [redacted] that day. We cannot simply multiply by 365 to see what the cost savings for a whole year would be. There are a few important remarks:

- **Monday and Friday have the most customer-orders.** The two days that are analyzed are a Monday and a Friday. These days are Picnic's busiest days. Consequently more tote-reduction can be achieved on these days than on other days. The data set used, contained [redacted] customer-orders, hence [redacted] each day. We should find out what the forecast tells us about the average customer-order size and the average amount of customer-orders a day in 2019. Picnic's growth team provided the required data.
  For further calculation we assume that the average size of a customer-order in the data set used is representative for the size of an average customer-order each day of that week. Since the tote-reduction depends on the size of a customer-order, we may then assume that the [redacted]% tote-reduction will be achieved every day. Note that the assumption of the customer-order size may influence the calculation.

- **Increasing customer-order size.** Picnic aims to encourage customers to place larger orders. At the end of 2019 the customer-orders are expected to be [redacted]% larger. On average the customer-orders will thus be approximately [redacted]% larger than the customer-orders in the data set that is used. In order to see what would happen with the amount of totes per customer-order given the same data-set with [redacted]% larger we reduced the volume capacity of the totes. Re-running the current tote-calculation and the *final heuristic* with [redacted]% smaller totes, gives us a reduction of [redacted] totes that day.

- **Increasing amount of customer-orders.** Picnic is growing rapidly. The growth team provided the data that gives the expectation of the amount of customer-orders per week in 2019. Taking the average of each week in 2019 and divid-

ing it by 7 gives us an average of [redacted] customer-orders each day. This means that the amount of customer-orders per day will increase with a factor [redacted], leading to an average reduction of [redacted] totes each day in 2019.

This brings us to the final calculation. By implementing the new tote-calculation described in Section 6.2 Picnic could save [redacted] in the year 2019.

7

Conclusions and Recommendations for further research

The main goal of this thesis, was to write an algorithm that finds good solutions to Picnic's bin-packing problem, by modelling several algorithms. Since the order in which the algorithms assign items to a tote has a major influence on the output of the algorithms, several item-orders are applied to the algorithms. Because ordering the items randomly gives remarkably good solutions, a new method is introduced where each customer-order is calculated at most eight times, each time shuffling the items before rerunning the algorithm, and remembering the best solution. It turned out that this method is optimal for [redacted]% of the customer-orders.

7.1. Recommendations

An analysis on the financial impact (Section 6.3) showed us that implementing the new heuristic can save Picnic [redacted] in 2019. This is based on a cost saving of [redacted] per reduced tote. It would be interesting to see if we can reduce the amount of totes used per customer-order even further. Also, given a certain amount of totes, some fulfillment processes could potentially be optimized by redistributing the items between the totes. This section is devoted to interesting topics for future research. A few are described, clustered in three main recommendations.

1. Adjust the fill-rate

Currently a fill-rate constraint is implemented in the tote calculation. A tote that is filled up to 85% of the volume capacity is considered full. This fill-rate is meant to manage several uncertainties:

- Differences in packaging types
- Measurement inaccuracy (see Appendix A.2)
- Untidiness during the packing process

Delivering totes filled with items that could have been packed in less totes is obviously a waste of money. On the other hand, if totes are too full to dispatch, it will cost extra time and thus money to rearrange the items such that putting the totes in the dispatch frames is possible. A right balance (fill-rate) needs to be found between too full and too empty totes. In the Python tote-calculation script, the current fill-rate of 85% is implemented. This percentage can be easily adjusted to calculate how many totes will be saved by increasing the fill-rate. This resulted in the percentages shown in Figure 7.1. The calculation in the financial impact

Figure 7.1: The percentage of totes saved when increasing the fill-rate, compared to the current fill-rate of 85%.

analysis in Section 6.3 can be applied to these percentages, resulting in a yearly cost-saving shown in Figure 7.2.

Some short practical research in one of Picnic's Fulfillment Centers has been done. Let us define a "full" tote as a tote that needs to be rearranged before dispatching, and a "too full" tote as a tote that cannot be dispatched at all. According to the FC-lead approximately 1 out of 20 totes are full. On the other hand, an employee that has been working for more than a year as full-time dispatcher said that he had never seen a "too full" tote. Given the high numbers shown in Figure 7.1 and Figure 7.2, I recommend analyzing the fill-rate and adjusting it if possible.

Figure 7.2: The yearly cost savings in €1000 when increasing the fill-rate.

2. Item to tote optimization

Without increasing the number of totes, items can be redistributed between totes to improve ergonomics and productivity. At the moment, items are not redistributed after the tote calculation is performed, which results in customer-orders with very heavy and full totes, as well as very light and empty totes. This distribution occurs because items are always placed in the first feasible tote, see Subsection 5.1.2. I recommend analyzing the following characteristics and implementing if possible:

- **Volume redistribution**
  The volume of a tote has a large effect on **pick productivity**. As explained in Recommendation 1, the items in full totes potentially need to be rearranged before dispatching. This occurs after a pick round, and takes about 30 seconds. Distributing volume will increase the pick productivity. A short calculation of the financial impact of redistributing volume follows.

  According to Picnic FC-lead, [redacted] need to be rearranged before dispatching. After running the tote-calculation, we can order the totes

in decreasing volume. Let us assume that the totes that need to be rearranged are the top [redacted]%, which is [redacted] totes in the two-day data set that we use. Note that redistributing items is only possible in multiple tote customer-orders.

Consider the following method to redistribute items. Let V_{70} be 70% of the volume capacity. First check if the summed volume of the items in the fullest tote is over V_{70}. If this is the case, take items one-by-one from the fullest tote and place it in the emptiest tote. Repeat this, until the summed volume of the items in the fullest tote is less than or equal to V_{70}. In the data set [redacted] out of the [redacted] full totes can be rearranged. [redacted] We can expand these two-day costs to the costs on a yearly basis the same way as described in the financial impact analysis in Section 6.3. This gives us a cost-saving of [redacted] for the year 2019.

- **Weight redistribution**
  Redistributing the weight will increase the **dispatch ergonomics**. The maximum weight of a tote is 20 kg. This is recorded as very heavy. When redistributing volume, the weight will probably be redistributed as well. If the method of redistributing items described above selects the heaviest items in the fullest totes and replaces them, we kill two birds with one stone.

- **Aisle skipping**
  The pick locations can be of major importance on **pick productivity**. The items in a tote determine which aisles must be passed. Currently, in a pick round always all aisles are passed, see Section 4.1. However, if the items of a customer-order can be clustered into the items of the picking aisles, potentially aisles can be skipped.

3. Tote to pick cart optimization

Picking the items is the highest labor intensive process in Picnic's Fulfillment Centers. The selection of totes on a pick cart is currently solely determined by the location in the dispatch frames. However, selecting totes based on other aspects could decrease the total walking distance of a shopper. All the following characteristics could increase **pick ergonomics and pick productivity**. It is therefore recommended to analyze them and if possible, implement them in the tote-calculation.

- **Number of different items**
  Customers often order larger quantities of the same item at once. An *order-line* is a single item, in a certain quantity. For each order-line, the shopper needs to walk from the pick cart to the aisle and back. The number of different order-lines determines the number of times a pick must be performed and should be considered when determining the position of the tote on a pick cart. Placing totes containing the same order-lines on a pick cart can increase the productivity.

- **Cross aisle travelling**
  Each aisle of the picking path has two sides. Placing totes on a pick cart in a way that minimizes cross aisle travelling will decrease the total walking distance.

- **Most items at front of pick cart**
  A pick cart is approximately 2.5 meters long and pulled from the front. Whilst picking, the bigger the distance between a tote and the front of the pick cart, the further the shopper needs to walk. Placing the totes with most different order-lines in the front of the pick cart would decrease the total walking distance.

Bibliography

[1] Retailtrends.nl. Het online aandeel berekend: 91 procent van de retailomzet is fysiek https://retailtrends.nl/item/54904/het-online-aandeel-berekend-91-procent-van-de-retailomzet-is-fysiek, 2018. [Online; accessed December 2018]

[2] Accountancyvanmorgen.nl. Omzet supermarkten stijgt in 2018 https://www.accountancyvanmorgen.nl/2018/06/21/omzet-supermarkten-stijgt-in-2018/, 2018. [Online; accessed December 2018]

[3] Distrifood.nl. Jumbo en Picnic winnen online van AH https://www.distrifood.nl/branche-bedrijf/nieuws/2018/11/jumbo-en-picnic-winnen-online-van-ah-101120059, 2018. [Online; accessed December 2018]

[4] Tatenhove van, J. An efficient algorithm for the VRPTW with short routes, Thesis for the Delft Institute of Applied Mathematics, 2016.

[5] Claymath.org. P vs NP Problem http://www.claymath.org/millennium-problems/p-vs-np-problem [Online; accessed September 2018]

[6] Cook, W.J., Cunningham, W.H., Pulleyblank, W.R., Schrijver, A. Combinatorial Optimization, 309-321, 1997.

[7] Complexity Labs. https://www.youtube.com/channel/UCutCcajxhR33k9URDdLsAQ/videos [Online video; accessed November 2018]

[8] Cambridge University. https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/turing-machine/one.html [Online; accessed November 2018]

[9] University of Cambridge Computer Laboratory. Section 4: Turing Machine Example Programs https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/turing-machine/four.html [Online; accessed October 2018]

[10] Cook, S.A., The complexity of theorem proving procedures, 151-158, 1971.

[11] Garey, M.R., and Johnson, D.S. Complexity results for multiprocessor scheduling under resource constraints. SIAM Journal on Computing 4, 397–411, 1975.

[12] Korte, B., Vygen, J. Combinatorial Optimization - Theory and Algorithms, 343-357, 363-264, 425-427, 2006.

[13] Papadimitriou, C.H., and Steiglitz, K. Combinatorial Optimization; Algorithms and Complexity. Prentice-Hall, Englewood Cliffs 1982, Chapter 11, 1982.

[14] Coffman, E.G., Jr., Garey, M.R., and Johnson, D.S. Approximation Algorithms for Bin-Packing – An Updated Survey. In Algorithm Design for Computer System Design, ed. by Ausiello, Lucertini, and Serafini. Springer-Verlag, 1984.

[15] Johnson, D.S. Fast Algorithms for Bin Packing. Journal of Computer and System Sciences 8, 272-314 (1974).

[16] Jungnickel D., The Greedy Algorithm. In: Graphs, Networks and Algorithms. Algorithms and Computation in Mathematics, vol 5. Springer, Berlin, Heidelberg, 1999.

[17] Fukunaga, A.S., Korf, R.E. Bin Completion Algorithms for Multicontainer Packing, Knapsack, and Covering Problems, 393-411, 2007.

[18] Martello, S., Toth, P. An upper bound for the zero-one knapsack problem and a branch and bound algorithm, 393-406, 1980.

[19] Christofides, N., Mingozzi, A., Toth, P. Loading problems. John Wiley Sons - Combinatorial Optimization, 1979.

[20] Land, A. H., Doig, A. G. An automatic method of solving discrete programming problems. 497-520, 1960.

[21] Schreiber, E.L., Korf, R.E. Improved bin completion for optimal bin packing and number partitioning. 651-657, 2013.

[22] Coffman, E.G., Jr., Garey, M.R., Johnson, D.S., and Tarjan, R.E. Performance bounds for level-oriented two-dimensional packing algorithms. SIAM Journal on Computing, 9:801–826, 1980.

[23] D.E. Knuth. The art of computer programming: A draft of sections 7.2.1.4-5: Generating all partitions. *Stanford University*: 1-4, 1986.

[24] Wolsey, L.A. Integer programming. A Wiley-Interscience Publication, 3-17, 1998.

[25] Gu, Z., Rothberg, E., Bixby, R., 2011. Gurobi Optimizer, Version 4.5.1. (Software program).

A

Appendix

A.1. The bin-packing problem and its complexity - extended version

This chapter is the extended version of Chapter 2. It is devoted to complexity theory. Some basic theory of the complexity classes is given in the first section. Then in the second section it is explained how a Turing machine works. We need this knowledge for the foundation of all *NP*-complete proofs: Cook's Theorem [10]. In section 3 the complexity of bin-packing will eventually be given and proven, by first proving the complexity of two other problems. It is important to know that decision problem version of the bin-packing problem is in the complexity class *NP*-complete, because then we know that there does not exist an algorithm that solves it in polynomial time, unless $P = NP$. For this reason we aim to find an algorithm in this thesis that outputs an approximation of a solution with an acceptable small error within reasonable time.

A.1.1. Problem complexity

Before introducing computational complexity, it should be clear what a decision problem is. A *decision problem* is a problem that, given an input, asks a yes-no question. Some examples:

1. Input: $x \in \mathbb{Z}$. Decision problem: "Is x a prime number?"

2. Input: a 9×9 matrix, partly filled in with integers 1,...,9. Decision problem: "Is there a Sudoku solution for the given matrix?"

3. Input: a set of items $1, ..., n$ with weights $w_1, ..., w_n$ and m bins with capacity C. Decision problem: "Can we pack items $1, ..., n$ into m bins?"

If the answer is yes, we call a corresponding solution a *yes-instance*. Later on in this chapter we need a more formal way to define a decision problem. A *decision*

problem is a pair $D = (X, Y)$, where X are all *instances* of D and $Y \subseteq X$ are all *yes-instances* of D. Taking the first example again, then $X = \{x : x \in \mathbb{Z}\}$ and $Y = \{x : x \text{ is prime}\}$. There are two questions that indicate the complexity of the decision problem:

1. Can we *solve* the decision problem within a reasonable time?
2. Given a yes-instance and a solution to this yes-instance. Can we *verify* the solution in reasonable time?

Here *within a reasonable time* means in polynomial time.

Definition 10 (Polynomial time)**.** *Assume we have a decision problem with an input of size n. An algorithm solves the problem in **polynomial time** if for some integers $C, k > 0$, it solves the problem in at most Cn^k steps.*

If a decision problem can be *solved* in polynomial time, the decision problem is in the complexity class *P*. If the yes-output of a decision problem can be *verified* in polynomial time, the decision problem is in the complexity class *NP*. Logically, $P \subseteq NP$.

The Clay Mathematics Institute of Cambridge awards a prize of $1 million to solve the next question [5]. If the solution to a problem can be verified in polynomial time, is it then always also solvable in polynomial time? This problem is better known as the *P versus NP problem*.

- If $P = NP$: All problems that can be verified in polynomial time, are also solvable in polynomial time. Decision problems that are not solvable in polynomial time, are called *NP*-hard.
- If $P \neq NP$: There are decision problems that can be verified in polynomial time, but cannot be solved in polynomial time. Decision problems that are not solvable in polynomial time, are called *NP*-hard. If a decision problem is not solvable in polynomial time but the answer is verifiable in polynomial time, it is called *NP*-Complete.

In Figure A.1 a simplistic Euler diagram is given that illustrates the sets. An often used definition of an *NP*-hard problem is that it is at least as hard as any other problem that is known to be *NP*-hard. Later on in this chapter we will prove that the bin-packing problem, formulated as the decision problem in the third example, is *NP*-complete. First we will give an often used definition of an *NP*-complete problem, for what we need to know what *polynomially reduce* means. Consider two decision problems D_1 and D_2. We say that D_1 *polynomially reduces* to D_2 if we can create a function that in polynomial time outputs a yes-instance of D_2 if and only if its input is a yes-instance of D_1.

Definition 11 (NP-complete)**.** *A decision problem $D \in NP$ is called **NP-complete** if all other problems in NP polynomially reduce to D.*

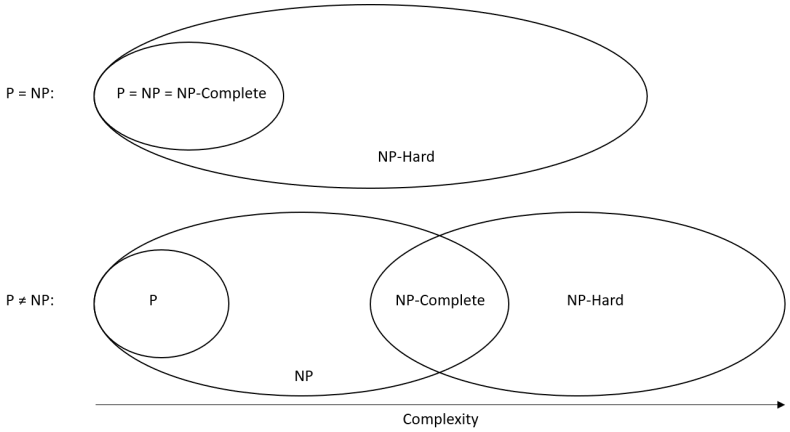


Figure A.1: *P* versus *NP*. Note that if *P=NP*, then *P=NP=NP*-complete.

It might now be clear that, to prove that bin-packing is *NP*-complete, we need to show that bin-packing is in *NP* ánd that all other problems in *NP* polynomially reduce to bin-packing.

A.1.2. Turing machine

To prove the complexity of the bin-packing problem, we need to know more about how a yes-instance can be verified. Alan Turing invented a machine that can verify instances: the Turing machine. Before we jump into the details of the Turing machine, we need a definition.

Definition 12 (Alphabet)**.** *A finite set A, consisting of at least two elements with no blanks is called an **alphabet**. The set of all possible strings consisting of elements in A of length at most n is denoted by* $A^* := \bigcup_{n\in\mathbb{N}} A^n$*. A **language** over A is a subset of* A^**. The elements of a language are **words**. The **length** of a string* $x \in A^n$ *is denoted by* $size(x) := n$*.*

Let us take an example that explains what an *alphabet* is. As *alphabet* we take $A := \{0,1,2,3,4,5,6,7,8,9\}$. As *language* we take all prime numbers up to 999. A *word* then is for example 173. As said in the introduction, the question whether a number is prime is a decision problem, $D = (X, Y)$. In this particular example, the set $X = \bigcup_{n=0}^{3} A^n$ (all natural numbers up to 999) and $Y \subseteq X$ exists of all prime numbers up to 999.

Assume we want to solve a certain decision problem. An algorithm is a way of writing down some very specific rules in such a way that if you follow them, it will lead you to the answer of your problem. Alan Turing came up with a machine that can solve algorithms. To do this, the input needs to be written as a string of symbols of an alphabet (see Definition 12) together with ⊔'s (blanks). At any moment in time the machine is in a particular state and it's looking at a value. The state the machine is in tells it what to do. The machine is programmed to do 3 things with this state and value: it overwrites the value, it moves to another position and it

goes into a (new) state. The machine reduces the string one by one to a new string containing elements of the alphabet with ⊔'s and eventually when it has reached its final state, the new string is the answer to your problem.

Definition 13 (Turing machine)**.** *Assume we have an alphabet A. Define $\overline{A} := A \cup \{\sqcup\}$. A **Turing machine** is defined by a function*

$$\Phi : \{0, ..., N\} \times \overline{A} \longmapsto \{-1, ..., N\} \times \overline{A} \times \{-1, 0, 1\}, \text{ for some } N \in \mathbb{N}.$$

Let us analyze this definition. We can see the Turing machine as the following function:

$$\Phi : (Current\ State,\ Symbol) \longmapsto (New\ State,\ New\ Symbol,\ Movement\ instruction).$$

This function sends 2-dimensional information to information in a 3-dimensional space. Given a current state (1) and a symbol (2), the Turing machine will go into a new state (1), overwrites the symbol (2) and moves to a new position (3).

Note that the set of new states has one more element than the set of current states, namely -1. This state is the final state. If the Turing machine is in state -1, the program is finished.

Definition 14 (Compute)**.** *Consider two languages of an alphabet A: $L, D \subseteq A^*$. Assume we have a function $f : L \to D$. Let Φ be a Turing machine. If for an $l \in L$ the Turing machine gives the* output$(\Phi, l) = f(l)$*, we say that Φ **computes** f.*

An example now follows. Assume we have a binary number as input and we want to add 1 to the number. In this example we thus have the alphabet $A = \{0, 1\}$ and language $L \subseteq A^*$ where the elements of L are numbers written in binary code. Also, we have a function f that is defined by $f(l) : l + 1$. The following Turing machine Φ computes f:

$$\begin{aligned}
\Phi(0, 0) &= (0, 0, 1) \\
\Phi(0, 1) &= (0, 1, 1) \\
\Phi(0, \sqcup) &= (1, \sqcup, -1) \\
\Phi(1, 0) &= (-1, 1, 0) \\
\Phi(1, 1) &= (1, 0, -1) \\
\Phi(1, \sqcup) &= (-1, 1, 1)
\end{aligned}$$

The Turing machine can begin at *any* value in the binary code, as long as the first state it is in, is 0. State 0 makes sure the beginning position is the most right value of the binary number. If the Turing machine is in that position, it switches to State 1. In State 1 the adding of the number 1 is done. In Figure A.2 an illustration is given of the example Turing machine, for the binary number 59.

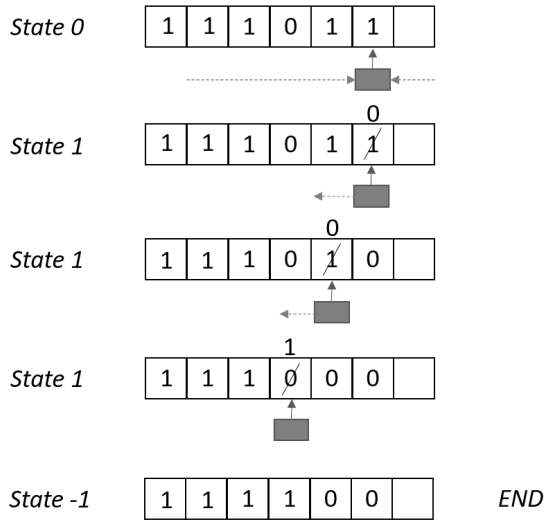


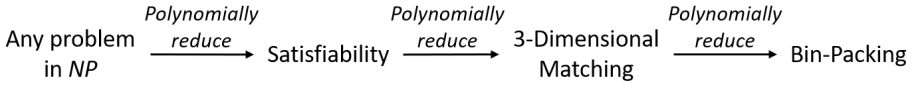
Figure A.2: Illustration of the Turing machine example where 1 is added to the binary number 59. The grey box represents the position of the Turing machine.

A.1.3. Complexity proof

The complexity of bin-packing will be proven in this section. To prove that a decision problem D is *NP*-complete, it should be proven that the problem is in *NP* and that any other problem in *NP* can be polynomially reduced to D. It is generally easy to see if a problem is in *NP*, but the second part is often a bit more complicated. The foundation of all *NP*-complete proofs lays in a theorem written and proven by Stephen Cook in 1971 [10]. It states that the SATISFIABILITY-problem is *NP*-complete.

Assume we have proven that SATISFIABILITY is *NP*-complete. Reducing an easy problem such as "Is an integer x dividable by 3?" to SATISFIABILITY does not say anything about the complexity of the easy problem. A problem could be made as difficult as one wants. A polynomial reduction the other way around does say something though. Assume we have a decision problem D. If we can polynomially reduce SATISFIABILITY to D, it means that D is at least as hard as SATISFIABILITY and therefore *NP*-hard. That is exactly what we will do in this section. The proofs in this section are based on proofs written by Bernhard Korte and Jens Vygen [12]

In the first subsection the SATISFIABILITY-problem will be explained. Furthermore, it will be proven that the problem is *NP*-complete. In the second subsection another decision problem will be introduced, namely the *3-Dimensional Matching*. Through this problem we will eventually be able to prove the complexity of bin packing:



Satisfiability

The first step in the triple proof will be to prove that SATISFIABILITY is *NP*-complete. The question that needs to be answered for a SATISFIABILITY-problem is, given a collection of clauses over a set of literals, is there a truth assignment such that the Boolean function corresponding to the family of clauses is true?

Definition 15 (Clause)**.** *Assume $X = \{x_1, ..., x_k\}$ is a set of Boolean variables and let $\overline{x}$ be the negation of x. Call the elements of the set $L := X \cup \{\overline{x} : x \in X\}$ **literals**. A **clause** over X is a subset of literals.*

Definition 16 (Satisfiable)**.** *A clause is true or **satisfied** if and only if at least one of its literals is true. A collection C of clauses over X is **satisfiable** if and only if there exists a truth assignment satisfying all of its clauses.*

Let us take an example that explains the SATISFIABILITY-problem. Assume we have $X = \{x_1, x_2, x_3\}$. Then the literals are $L = \{x_1, x_2, x_3, \overline{x_1}, \overline{x_2}, \overline{x_3}\}$. If we now take a collection of clauses $\{\{x_1, x_3\}, \{\overline{x_1}, \overline{x_2}\}, \{\overline{x_1}, x_2, x_3\}\}$, then this corresponds to the Boolean function $(x_1 \wedge x_3) \vee (\overline{x_1} \wedge \overline{x_2}) \vee (\overline{x_1} \wedge x_2 \wedge x_3)$. This collection of clauses is satisfiable, because for example the assignment $(x_1, x_2, x_3) = (true, false, true)$ satisfies the problem. An easy example of a no-instance would be the collection of clauses $\{\{x_1, x_2\}, \{\overline{x_1}, \overline{x_2}\}\}$. No assignment can be found such that the Boolean function $(x_1 \wedge x_2) \vee (\overline{x_1} \wedge \overline{x_2})$ is true, since $(\overline{x_1} \wedge \overline{x_2})$ is the negation of $(x_1 \wedge x_2)$ and therefore cannot both be true. Now that the SATISFIABILITY-problem is explained, we need only one more definition before we can start the proof.

Definition 17 (Certificate)**.** *Let $D = (X, Y)$ be a decision problem. D is in* NP *if there exists a corresponding problem $D' = (X', Y')$ in P and a polynomial p with*

$$X' := \{x\#c : x \in X, c \in \{0,1\}^{\lfloor p(size(x)) \rfloor}\},$$

such that

$$Y = \{y \in X : \exists\, c \in \{0,1\}^{\lfloor p(size(x)) \rfloor} \text{ with } y\#c \in Y'\},$$

*where $x\#c$ is a string containing x, the symbol # and c. If there exists a string c with $x\#c \in Y'$, we say that c is a **certificate** for x. An algorithm that checks if an instance of D' is a yes-instance, is called a **certificate-checking algorithm**.*

Theorem 4 (Cook, 1971)**.** *Satisfiability is NP-complete.*

Proof idea. We take a problem D in *NP* that is not SATISFIABILITY and polynomially reduce it into SATISFIABILITY. Since $D = (X, Y)$ is in *NP*, it is verifiable in polynomial time. Furthermore, by Definition 17 there exists a polynomial-time Turing machine Φ for a corresponding problem $D' = (X', Y')$, polynomially depending on $size(x\#c)$. We will define an upper bound on the computationlength of Φ, for any input $x\#c \in X'$ and we will define a set of Boolean variables $V(x)$. We will construct a collection of clauses $C(x)$ over $V(x)$ in such a way that $C(x)$ is satisfiable if and only if x is a yes-instance of D ($x \in Y$). To prove this "if and only if relation", we

again need Definition 17. Since Φ is a certificate checking algorithm, we will conclude that if the output of the Turing machine is 1, $x \in Y$. Conversely, if $x \in Y$, we show that indeed $C(x)$ is satisfiable.

Proof. It is clear that SATISFIABILITY belongs to *NP*. We must prove that it is *NP*-complete by showing that any decision problem in *NP* can be polynomially reduced to SATISFIABILITY (Definition 3). Let $D = (X, Y)$ be a problem in *NP* that is not SATISFIABILITY. Let $D'(X', Y')$ be the corresponding decision problem from Definition 17 and let Φ be a polynomial-time Turing machine for D' with alphabet A with $\overline{A} = A \cup \sqcup$. There is a polynomial b depending on $size(x\#c)$ such that $\text{time}(\Phi, x\#c) \leq b(size(x\#c))\ \forall\ x\#c \in X'$. Define $l(x) := \lfloor p(size(x)) \rfloor$. Let $B := b(size(x) + 1 + l(x))$ be an upper bound on the computation length of Φ on input $x\#c$, $\forall\ c \in \{0,1\}^{l(x)}$.

We will construct a collection of clauses $C(x)$ such that $C(x)$ is satisfiable if and only if x is a yes-instance ($x \in Y$). First we define the set $V(x)$ of Boolean variables for each $x \in X$:

- $v_{i,j,\sigma}$ for all $0 \leq i \leq B, -B \leq j \leq B$ and $\sigma \in \overline{A}$: indicates whether at time i the j-th position of the string contains the symbol σ;
- $w_{i,j,n}$ for all $0 \leq i \leq B, -B \leq j \leq B$ and $-1 \leq n \leq N$: indicates whether at time i the j-th position is scanned and the instruction in state n is executed.

We will now construct the collection of clauses $C(x)$ over the set $V(x)$. There is a symbol in each position at any time:

- At time i and position j, the string contains a symbol $\sigma \in \overline{A}$:
  $\{v_{i,j,\sigma} : \sigma \in \overline{A}\}$ for $0 \leq i \leq B$ and $-B \leq j \leq B$;
- Let $\sigma, \tau \in \overline{A}$ with $\sigma \neq \tau$. At time i and position j, the string can only contain the symbol σ if it does not contain the symbol τ:
  $\{\overline{v_{i,j,\sigma}}, \overline{v_{i,j,\tau}}\}$ for all $0 \leq i \leq B, -B \leq j \leq B$.

There is a single instruction executed in a unique position at any time:

- At time i a symbol is scanned at position j in state n:
  $\{w_{i,j,n} : -B \leq j \leq B, -1 \leq n \leq N\}$ for $0 \leq i \leq B$;
- Let $-B \leq j, k \leq B$ and $-1 \leq n, m \leq N$ with $(j, n) \neq (k, m)$. At time i, position j can only be scanned in state n if position k is not scanned in state m:
  $\{\overline{w_{i,j,n}}, \overline{w_{i,k,m}}\}$ for $0 \leq i \leq B, -B \leq j$.

The algorithm starts correctly:

- At time 0 and position j, the string contains the j-th symbol of x:
  $\{v_{0,j,x_j}\}$ for $1 \leq j \leq size(x)$;
- At time 0 and position $size(x) + 1$, the string contains the symbol #:
  $\{v_{0,size(x)+1,\#}\}$;
- At time 0 and position $size(x) + 1 + j$, the string contains either a 0 or a 1:
  $\{v_{0,size(x)+1+j,0}, v_{0,size(x)+1+j,1}\}$ for $1 \leq j \leq l(x)$;

- At time 0 and position j, the string contains the symbol $\sqcup$:
  $\{v_{0,size(x)+1,\sqcup}\}$ for $-B \leq j \leq 0$ and $size(x)+2+l(x) \leq j \leq B$;
- At time 0 and position j, the instruction state is 0:
  $\{w_{0,1,0}\}$.

The algorithm works correctly:

- Assume $\Phi(n,\sigma) = (m,\tau,\delta)$. Suppose the algorithm is in time i, the position it is in is j and the symbol that is scanned is σ in the state n.
  - Then the symbol σ is changed into the symbol τ:
    $\{\overline{v_{ij\sigma}}, \overline{w_{ijn}}, w_{i+1,j,\tau}\}$ for $0 \leq i < B, -B \leq j \leq B, \sigma \in \overline{A}$ and $0 \leq n \leq N$.
  - Then the state will be m in position $j+\delta$:
    $\{\overline{v_{ij\sigma}}, \overline{w_{ijn}}, w_{i+1,j+\delta,m}\}$ for $0 \leq i < B, -B \leq j \leq B, \sigma \in \overline{A}$ and $0 \leq n \leq N$.
- When the algorithm reaches state -1, it stops:
  - If in time i the position is j and the state is -1, the state will stay -1 in position j:
    $\{\overline{w_{i,j,-1}}, w_{i+1,j,-1}\}$ for $0 \leq i < B, -B \leq i \leq B$ and $\sigma \in \overline{A}$;
  - If in time i the position is j, the state is -1 and the symbol σ is scanned, the symbol will stay σ in position j:
    $\{\overline{w_{i,j,-1}}, \overline{v_{i,j,\sigma}}, v_{i+1,j,\sigma}\}$ for $0 \leq i < B, -B \leq i \leq B$ and $\sigma \in \overline{A}$.
- Symbols in positions that are not scanned do not change:
  - Assume at time i, the symbol in position j is σ. If at time i the symbol in position k is scanned, the symbol σ in position j will stay σ:
    $\{\overline{v_{i,j,\sigma}}, \overline{w_{i,k,n}}, v_{i+1,j,\sigma}\}$ for $0 \leq i \leq B, \sigma \in \overline{A}, -1 \leq n \leq N$ and $-B \leq j,k \leq B$ with $j \neq k$.
- The output of the algorithm is 1:
  - At the end, at time B the symbol in position 1 is 1:
    $\{v_{B,1,1}\}$;
  - At the end, the symbol in position 2 is $\sqcup$:
    $\{v_{B,2,\sqcup}\}$.

Now it remains to show that $C(x)$ is satisfiable if and only if $x \in Y$.
" $\Longrightarrow$ " Let $x \in X$ and assume $C(x)$ is satisfiable with a satisfying truth assignment T. We will show that $x \in Y$ by giving a certificate c for x. Let $c \in l(x)$ with $c_j = 0 \; \forall \; j$ with $T(v_{0,size(x)+1+j,0})$=TRUE and $c_j = 1$ otherwise. The construction above represents the computation of Φ on input $x\#c$. Since $C(x)$ is satisfiable, the output of the algorithm is output$(\Phi, x\#c) = 1$. By Definition 17 it now follows that $x \in Y$.
" $\Longleftarrow$ " Assume $x \in Y$ and let c be a certificate for x.

Now consider the computation of the the Turing machine Φ on input $x\#c$. Define $T(v_{i,j,\sigma})$=TRUE if and only if at time i, in position j the symbol is σ. Also

define $T(w_{i,j,n})$=TRUE if and only if at time i, the symbol that is scanned is in position j and the state it is in is n. Furthermore, if the computation is done, we want the symbols to stay the way they are: for $i = m+1,...,B$ set $T(v_{i,j,}) := T(v_{i-1,j,\sigma})$ and $T(w_{i,j,n}) := T(w_{i-1,j,n})$ $\forall$ j,n and σ. Now T is a truth assignment satisfying $C(x)$. ■

3-Dimensional Matching

The next step is to prove that 3-Dimensional Matching (3DM) is *NP*-complete, by reducing SATISFIABILITY to 3DM. The question that needs to be solved for a 3DM-problem is, given 3 disjoint sets and edges that match one element from each set, can we find a subset of these edges such that all elements are matched exactly once? In Figure A.3 two examples are given that illustrate the following definition.

Definition 18 (3-Dimensional Matching)**.** *Given 3 disjoint sets X, Y, Z of equal cardinality and a set of edges $E \subseteq X \times Y \times Z$. Is there a subset $S \subseteq E$ such that $|S| = |X|$ and for each pair $(x,y,z),(x',y',z') \in S$ one has $x \neq x'$, $y \neq y'$ and $z \neq z'$?*

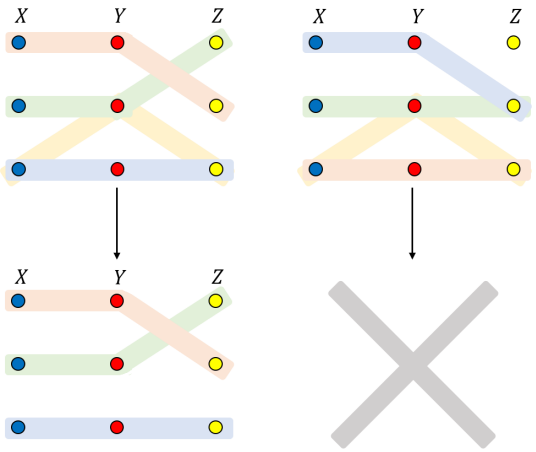


Figure A.3: Two examples of a 3DM instance. The left one has a solution, the right one does not.

Theorem 5 (Karp, 1972)**.** *3DM is NP-complete.*

Proof. Obviously 3DM is in *NP*. We need to prove 3DM is *NP-hard*. We will do this by polynomially reducing SATISFIABILITY to 3DM. Consider a collection C of clauses $C_1,...,C_m$ over $X = \{x_1,...,x_n\}$. We will construct an instance (X,Y,Z,T) of 3DM in such a way that it is a yes-instance if and only if C is satisfiable. Define the 3 disjoint sets, all with cardinality $2nm$:

$$X := \{x_i^j, \overline{x_i}^j : i = 1,...,n; j = 1,...,m\}$$
$$Y := \{y_i^j : i = 1,...,n; j = 1,...,m\} \cup \{yy^j : j = 1,...,m\} \cup \{yyy_k^j : k = 1,...,n-1; j = 1,...,m\}$$
$$Z := \{z_i^j : i = 1,...,n; j = 1,...,m\} \cup \{zz^j : j = 1,...,m\} \cup \{zzz_k^j : k = 1,...,n-1; j = 1,...,m\}$$

Now define all matching edges E:

$E_{1a} := \{(x_i^j, y_i^j, z_i^j) : i = 1,...,n; j = 1,...,m\}$

$E_{1b} := \{(\overline{x_i}^j, y_i^j, z_i^a) : i = 1,...,n; j = 1,...,m; a = 1,...,m,$ where $y_i^{m+1} := y_i^1$ and $a = j$ unless $i = n$, then $a = 1\}$

$E_1 := E_{1a} \cup E_{1b}$

$E_{2a} := \{(x_i^j, yy^j, zz^j) : i = 1,...,n; j = 1,...,m; x_i \in C_j\}$

$E_{2b} := \{(\overline{x_i}^j, yy^j, zz^j) : i = 1,...,n; j = 1,...,m; \overline{x_i} \in C_j\}$

$E_2 := E_{2a} \cup E_{2b}$

$E_{3a} := \{(x_i^j, yyy_k^j, zzz_k^j) : i = 1,...,n; j = 1,...,m; k = 1,...,n-1\}$

$E_{3b} := \{(\overline{x_i}^j, yyy_k^j, zzz_k^j) : i = 1,...,n; j = 1,...,m; k = 1,...,n-1\}$

$E_3 := E_{3a} \cup E_{3b}$

$E := E_1 \cup E_2 \cup E_3$

Note that in E_2 the clauses of C are defined. C is satisfiable if and only if all clauses are true. Given the constructed edges and sets, we will now prove that (X, Y, Z, E) is a yes-instance if and only if it is a satisfying truth assignment.
" $\Longrightarrow$ " Assume (X, Y, Z, E) is a yes-instance and let $S \subseteq E$ be a solution. Since S is a solution, all elements in X, Y and Z must be matched by edges in S. An illustration is given in Figure A.4, with $m = 2$ and $n = 2$. E_1, E_2 and E_3 exist of 2 sets of matching edges, a and b. The set a is drawn as a continuous line, b as a dotted one. The

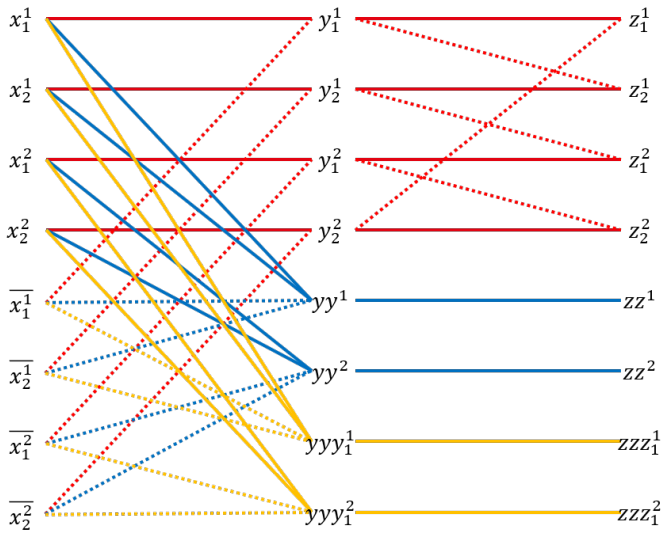


Figure A.4: Illustration of the 3DM instance used in the proof of Theorem 5.

red lines represent E_1, the blue lines E_2 and the yellow lines E_3. In the figure we can easily see that the elements y_i^j and z_i^j for all i and j are only matched by red

lines. Also, combining red dotted lines with red continuous lines is not possible in a solution. Hence we have 2 options:

1. $S \cap E_1 = E_{1a}$, i.e. all red *continuous* lines are in the solution S. Set x_i to false. Since yy^i still must be matched, only the blue *dotted* lines can be used. They are all connected to the $\overline{x_j}^i$, so the literals are both true. Hence we have a satisfying truth assignment.
2. $S \cap E_1 = E_{1b}$, i.e. all red *dotted* lines are in the solution S. Set x_i to true. Since yy^i still must be matched, only the *continuous* blue lines can be used, which are now connected to the true x_j^i, so again the literals are both true. Hence we have a satisfying truth assignment.

Hence, a 3DM yes-instance implies a satisfying truth assignment.
" $\Longleftarrow$ " Assume we have a satisfying truth assignment. Then the blue lines in the solution are either all dotted, or all continuous. So, there are 2 options:

1. $S \cap E_2 \subseteq E_{2a}$, i.e. a subset of the blue *continuous* lines match yy^j and zz^j for $j = 1, ..., m$. Define these lines a set $S_2 \subseteq S$. S_2 has cardinality m. The only option to match y_i^j and z_i^j for all i and j are all the lines in E_{1b}. We have a set $S_1 \subseteq S$ of cardinality nm.
2. $S \cap E_2 \subseteq E_{2b}$, i.e. a subset of the blue *dotted* lines match yy^j and zz^j for $j = 1, ..., m$. Define this as a set $S_2 \subseteq S$ of cardinality m. The only option to match y_i^j and z_i^j for all i and j now, are all the lines in E_{1b}. Again we have a set $S_1 \subseteq S$ of cardinality nm.

In both options we see that for each distinct $(x, y, z), (x', y', z') \in S_1 \cup S_2$ we have $x \neq x', y \neq y'$ and $z \neq z'$. Obviously we can add $(n-1)m$ elements from E_3 to $S_1 \cup S_2$, to complete a solution. Hence, a satisfying truth assignment implies a 3DM yes-instance. ■

Bin-packing

The proof of the following theorem is based on a proof written by Papadimitriou in 1982 [13].

Theorem 6. *Given an instance I of a bin-packing problem. It is NP-complete to decide whether I can be solved with B bins.*

Proof. Clearly the problem is in *NP*, because we can easily verify in polynomial time that a given partition is feasible. It leaves us to show that all other problems in *NP* polynomially reduce to this decision problem. We shall reduce 3-Dimensional matching to it.

Given the sets of nodes $X = \{x_1, ..., x_n\}$; $Y = \{y_1, ..., y_n\}$ and $Z = \{z_1, ..., z_n\}$ and the set of triples $T = \{t_1, ..., t_m\} \subseteq X \times Y \times Z$. We are asked whether there is a set of n triples in T, such that each node in X, Y and Z is contained in exactly one of the n triples.

We will construct an instance of bin-packing that has $N = 4m$ items. Each triple in T corresponds to an item, denoted simply t_j. Furthermore, each occurrence

of a node in X, Y, and Z to a triple corresponds to an item. Let $u_i \in X \cup Y \cup Z$. Then u_i will be denoted by $u_i[1], u_i[2], ..., u_i[N(u_i)]$, where $N(u_i)$ is the number of occurrences of u_i in the triples. The sizes of the items are shown below:

$$\text{triple } (x_i, y_j, z_k) \in T \text{ has weight } 10M^4 + 8 - iM - jM^2 - kM^3$$

$$x_i[q] \text{ has weight } \begin{cases} 10M^4 + iM + 1 \text{ if } q = 1 \\ 11M^4 + iM + 1 \text{ if } q > 1 \end{cases}$$

$$y_j[q] \text{ has weight } \begin{cases} 10M^4 + jM^2 + 2 \text{ if } q = 1 \\ 11M^4 + jM^2 + 2 \text{ if } q > 1 \end{cases}$$

$$z_k[q] \text{ has weight } \begin{cases} 10M^4 + kM^3 + 4 \text{ if } q = 1 \\ 8M^4 + kM^3 + 4 \text{ if } q > 1 \end{cases}$$

Here, M is a very large number, say $100n$. Note that there is a difference in size between (arbitrarily) the first occurrence and the other occurrences of the nodes in X, Y and Z. Define the bin capacity $C := 40M^4 + 15$. This capacity makes it possible to fit exactly one triple and one node of all three sets X, Y and Z as long as the nodes are either all three or none of the three a first occurrence. There are m bins, as many as triples.

Assume all items fit into m bins. Note that the sum of all items is mC. Hence, each bin must be full. Also, note that the weight of each item is strictly between $\frac{1}{5}$ and $\frac{1}{3}$. Hence, each bin must contain four items. We have $C \mod M = 15$. Given are the numbers 1, 2, 4, and 8. Even if we allow repetition, there is only one way to get the number 15 by choosing four numbers: each number must be chosen once. Furthermore, the sum modulo M^2 must also be 15, so $(i' - i) \cdot M + 15 = 15 \mod M^2$, thus $i = i'$. Similarly, taking the sum modulo M^3 and M^4, get $j = j'$ and $k = k'$. Each bin thus contains a triple $t = (x_i, y_j, z_k)$, together with $x_i \in X$, $y_j \in Y$ and $z_k \in Z$. Furthermore, since $40M^4$ must be reached, either all three occurrences are first occurrences or none of them are. Hence, there are n bins that contain only first occurrences, the n triples in these bins form a 3-Dimensional matching.

Conversely, assume a 3-Dimensional matching exists. Making sure that the triples in the matching get first occurrences of all three members, we can fit all items into the m bins by matching each triple with occurrences of its members. ■

Picnic's bin-packing problem is two-dimensional (each item has a weight ánd a volume). Corollary 2 says that the decision problem version of the two-dimensional bin-packing problem is also *NP*-complete. It is therefore our goal to find an algorithm in this thesis that outputs an approximation of a solution with an acceptable small error instead of finding a solution, assuming $P \neq NP$.

Corollary 2. *Given an instance of a two-dimensional I bin-packing problem. It is NP-complete to decide whether I can be solved with B bins.*

Proof. Clearly it is in *NP* to decide whether I can be solved with B bins. It leaves us to show that it is *NP*-complete. Consider a bin-packing instance I_1 that is one-dimensional; each item has a volume and each bin has a volume capacity V. Adding

weight 0 to each item and adding the weight capacity W to each bin gives us an instance of a two-dimensional bin-packing problem. Obviously this is a yes-instance if and only if I_1 is a yes-instance. We see that one-dimensional bin-packing is a special case of two-dimensional bin-packing. We may conclude that it is *NP*-complete to decide whether a two-dimensional bin-packing problem instance can be solved with B bins. ■

A.2. Project for data accuracy

For a highly efficient supply chain, we need information about Picnic's products. To be able to run the algorithm for tote-calculation, the weight and volume is necessary. For the construction of the picking shelves, the packaging type and three product characteristics need to be known: fragility, stackability and contaminating. Picnic has purchased a 3D measurement device that should give accurate data. The data accuracy project consisted of four main tasks:

1. Writing measurement instructions
2. Training Picnic workers to do the measurements
3. Creating a data-validation system
4. Doing a fill-rate analysis on new data

The first, third and fourth tasks are written down below.

Task 1. Writing measurement instructions

Certain product characteristics need to be known to create an efficient logistical flow. These characteristics are *Fragility*, *Stackability* and *Contaminating*. To understand how to decide whether a product has a certain characteristic, it should be known exactly what is meant by the characteristic. After that, it can be decided how to write a decision instruction in such a way that everybody would assign the same characteristics to a certain product. I decided to create flow-charts. Each flow-chart is a decision tree where answering the questions will eventually lead to whether a product has a characteristic or not.

Fragility

Picnic wants their products to arrive in a perfect condition at their customers. Therefore they want fragile products to be picked at last, to make sure that they are at the top in a bag. These products should get the label *fragile*. Intuitively these are the products that you cannot lay under a 2L pack of milk. Note that fragility in different companies can mean something different. For example, a company that sells and delivers glasses would consider itself to handle fragile products. Since Picnic delivers their groceries in a hard plastic crate, glass bottles should not get the label *fragile*. Examples of products that should get the label *fragile* at Picnic are raspberries, eggs and chips. The flow-chart to decide whether a product is fragile is shown in Figure A.5:

As you might notice, the fragility flow-chart includes a subjective question, namely "Does the packaging break easily when a 2L pack milk is laid on it?". I recognize the subjectivity. After trying to change this, I noticed that the decision tree became harder to use. Discussing the problem with several Picnic people lead to the decision to keep the subjective question in the flow-chart.

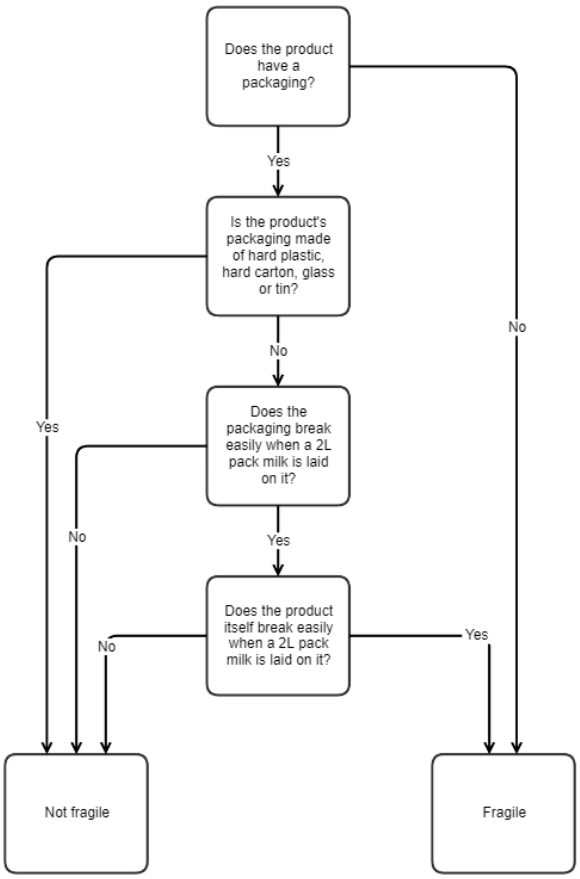


Figure A.5: Flowchart for the characteristic *Fragile*.

Stackability

In the distribution center, the shelf capacity should be used as much as possible. Therefore, it is important to know whether a product can be stacked. If so, several layers can be placed on top of each other, to reduce the needed shelf space. There are products that intuitively can be stacked, but in practice should not be stacked. For example products that have a glass packaging should never be stacked due to safety reasons. The flow-chart to decide whether a product can be stacked is shown in Figure A.6. There has been a discussion about the definition of a stackable product. It is clear that for example shoe boxes can be stacked, but what about bags of rice? Two layers are fine, but the more bags you will stack, the more the stack will look like a pyramid. I learned that Picnic plans the shelf filling such that only a few layers are used. Therefore a bag of rice is considered stackable.

Contaminating

Picnic collects a lot of data through the feedback it gets from customers. This way, it showed that some people dislike the idea of having a chemical product or animal food in the same bag as the food they eat. Therefore the characteristic *Contaminating* needs to be checked for every product. The flow-chart to decide whether a

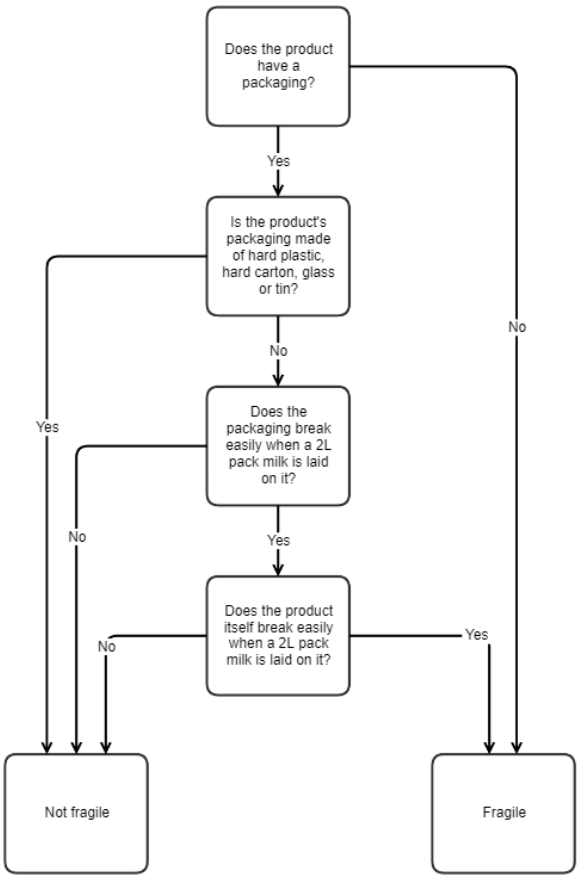


Figure A.6: Flowchart for the characteristic *Stackable*.

product is contaminating is shown in Figure A.7. Unlike the fragility flow-chart, it is easy to wright down in an objective way whether a product is contaminating or not. There are only three categories of products that should be labeled as contaminating: animal food in a soft packaging, chemicals that are not used for cleaning humans and barbecue products.

Measurement instruction

1. Grab the product and scan the barcode.
2. Take a picture of the ingredient list (only applicable for food products)
   (a) Keep the list in front of the camera
   (b) Make sure the lens is focussed on the list
   (c) Press the button with the camera symbol
3. Put the product on the plate in the following way:
   (a) Place it with the (scannable) barcode facing forward, unless:
      - It's a fluid (fluids always upright!)

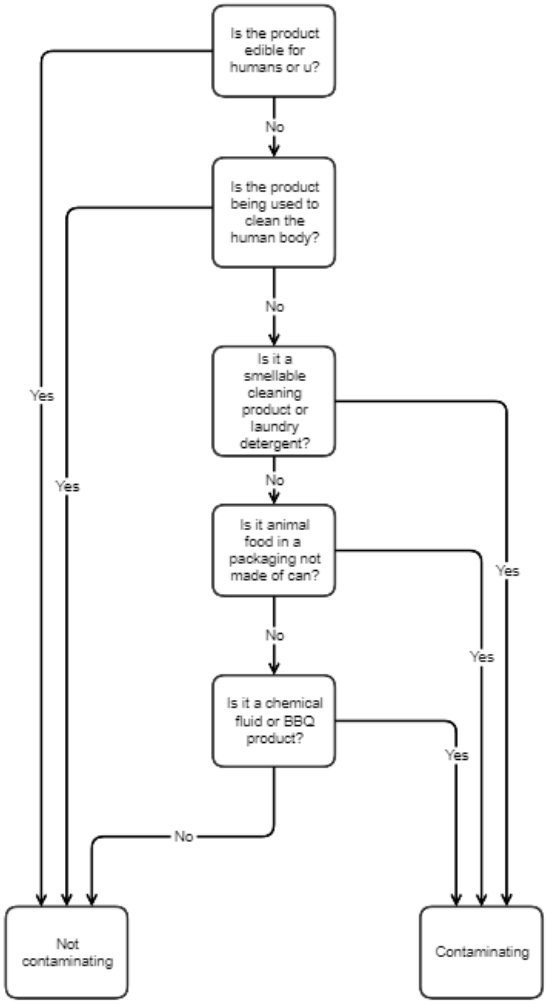


Figure A.7: Flowchart for the characteristic *Contaminating*.

- The barcode facing forward is an unstable way of placing the product
- The product is damaged if the barcode is facing forward (ready meals must always stand upright)

(b) Make sure the product is placed as straight as possible against the line.

4. Push “start”. The product will be measured now.
5. Fill in the fields Packaging type, Fragility, Contaminating and Stackability according to the corresponding flow-charts
6. Press “Save” and take the product off the plate
7. Check if all values below “Meetresultaten” are 0. If so, go to step 8.

(a) If not, press “kalibreren”. All values should be 1 now

- If not, send the supervisor a slack-message. Do not continue mea-

suring.

8. Put the product back and start with step 1 again with the next product.

Task 2. Creating a data-validation system

It is clear that the flow-charts can be interpreted differently by different people. Furthermore, if the device is not calibrated in the right way, fault data will be produced. Therefore data-validation needs to be done. There are two ways to validate the measurements. The first is to compare the measurement to the data that is known. Although the existing data is not very reliable, measurements should stand out if the total volume or weight differs much. Also fragility, contaminating and stackability are already known in the old database, so the new characteristic can be compared to the old data. The second way to validate a measurement is in the measurement itself. Cross-checks can be executed. For example, a bottle (packaging type) is never fragile (product characteristic).

1. Compare measurements to old data:
    - Volume and Weight
        - If the differences are over 20%, remeasure
    - Fragile/Stackable/Contaminating
        - If old and new data differ, check photo
        - Fill in yes or no according to flow-charts
2. Cross-check
    - If the following data occurs, check photo and fill in yes or no according to flow-charts:
        - (*Fragile*) and (Blister pack, Bottle, Crate, Bucket, Jar, Can, Tube, Pack, *Contaminating*)
        - (*Contaminating*) and (*Fragile*, Not packed, Pack, Vacuum Tray, Net)
        - (*Stackable*) and (Bottle, Tube)
        - (*Not stackable*) and (Bucket)

If a measurement did not make it through the validation, it was measured again another day. The data validation that is described, has been programmed in Python.

Task 4. Doing a fill-rate analysis on new data

As soon as the new data will be implemented, there will fit a different amount of products in a tote. On one hand this could mean that in the FC products will be stuffed in a tote. This could cause product quality loss. On the other hand it could mean that with the new data unnecessarily much air will be moved and thus make

unnecessary costs. In other words, a fill-rate analysis had to be done to make sure the logistical process of Picnic would not lose efficiency. From now on, the data Picnic used to use is called *Previous data* and we assume the measurements are correct, hence call them *Actual data*.

For this analysis a week of customer-orders is used. The current bin-packing algorithm is analyzed, to be able to simulate how many totes are used for one order. For this analysis, the interesting totes are the volume restricted totes. Therefore all volume restricted totes are filtered for the three temperature zones; ambient, chilled and frozen. It is compared how full they were according to the previous data with how full they actually were. Not only the average differences were analyzed, but also the distribution of the actual volumes if the previous volume was near to the maximum volume was analyzed.

Ambient
Looking at the totes that where volume restricted, we see the following:

Previous volume:	78.43%
Actual volume:	76.46%
Average decrease:	1.98%

In Figure A.8 you see a graph where the darker line shows how full Picnic thought a tote was, with a corresponding lighter line that shows the exact data. As expected

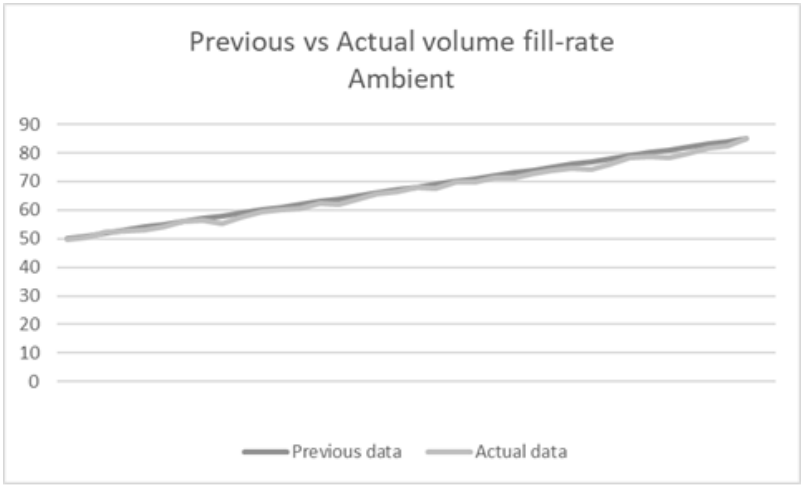


Figure A.8: Previous versus Actual volume fill-rate of ambient totes.

because of the average decrease of only 1.98%, we see a line that is very close to the original line. To see what the variation is of the actual volume versus the previous volume, a histogram follows with all the actual volumes corresponding to previous volumes 84% and 85% in Figure A.9. We see almost a normal distribution, but with a little bit more area to the left of the peak. This means that there have been more totes that were filled up to less than the maximum volume than the other

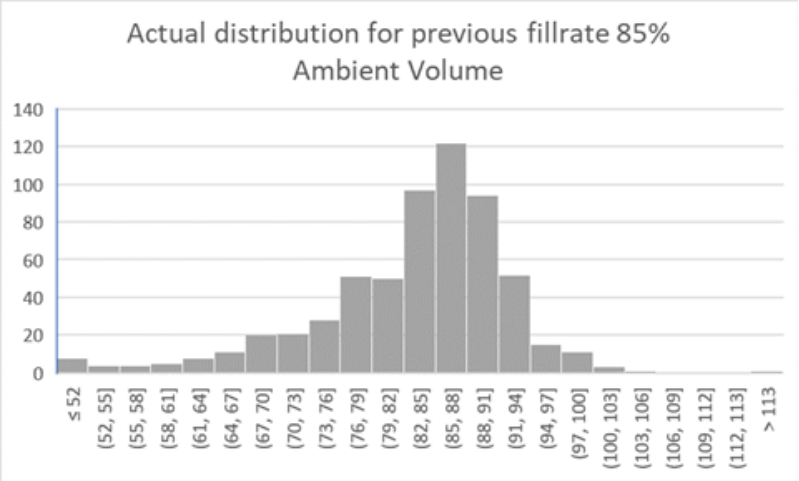


Figure A.9: Actual distribution for previous fill-rate of 85% of ambient totes.

way around. Combining this with the average volume decrease of 1.98%, implementing the actual data without adjusting the tote-calculation could cause more overfull totes.

Chilled

As the name suggests, in chilled totes products are packed that are cold. Therefore there are ice-packs packed into each tote. The tote-calculation does not take these into account. Looking at the chilled totes that where volume restricted, we see the following:

Previous volume:	81.29%
Actual volume:	72.44%
Average decrease:	8.85%

In Figure A.10, you see a graph where again the darker line shows how full Picnic thought a tote was, with a corresponding lighter line that shows the exact data.

As for the ambient totes, it was expected to see this lighter line under the darker line, but here we see that the difference between the previous- and actual volume differs more as the maximum volume is reached. The histogram with all the actual volumes corresponding to previous volumes 84% and 85% is shown in Figure A.11.

Again we see more area to the left of the peak. The same applies to the chilled area: not adjusting the tote-calculation could cause more overfull totes.

Frozen

As for the chilled totes, there are extra cooling products packed in the frozen totes. Also here we could say therefore that the the totes are actually fuller. Looking at the frozen totes that where volume restricted, we see the following:

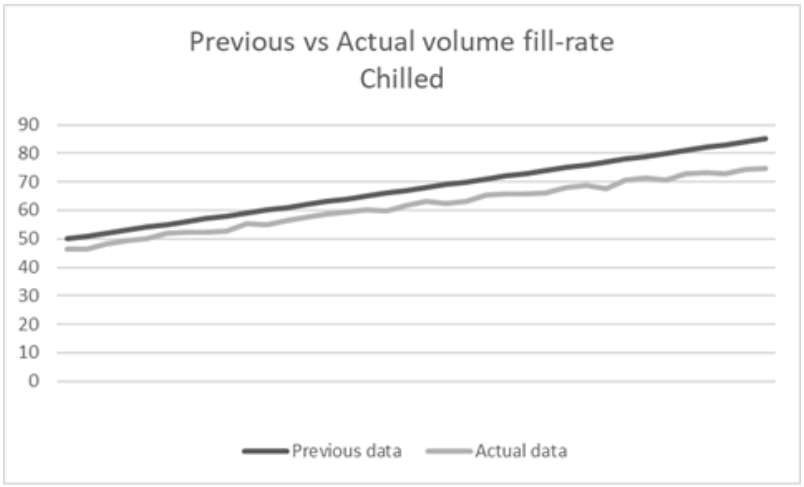


Figure A.10: Previous versus Actual volume fill-rate of chilled totes.

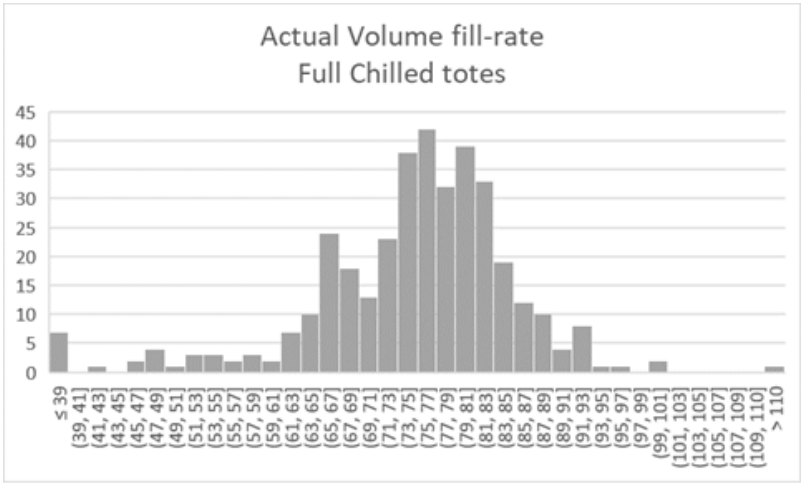


Figure A.11: Actual distribution for previous fill-rate of 85% of chilled totes.

Previous volume:	68.64%
Actual volume:	75.99%
Average increase:	7.35%

Note that unlike ambient and chilled, there is an average volume increase. Again, you see a graph where the darker line shows how full Picnic thought a tote was, with a corresponding lighter line that shows the exact data.

Normally, frozen products are ordered less than ambient and chilled products. Consequently, a frozen tote being volume restricted is rare. Therefore a histogram as shown for ambient and chilled with only the totes that are previously filled up to 84% and 85% does not give much information. For that reason the histogram is not shown here.

The increase in volume means that the totes have been filled up to a higher volume than previously thought.

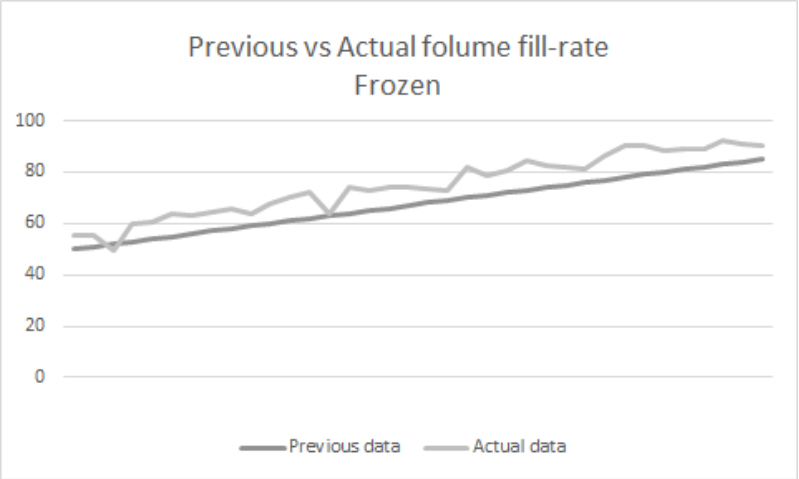


Figure A.12: Previous versus Actual volume fill-rate of frozen totes.

Recommandations

It is important to understand that because the actual ambient and chilled volume is less than the previous volume, there fit more products in one tote than previously thought. On one hand this is positive news since eventually less totes need to be used per customer-order. On the other hand, untidiness during the packing process can cause air between the products in the tote. The more products, the more air. Hence, the more products you can put in one tote, the lower the maximum volume used in the tote-calculation should be.

Currently the the totes are filled up to 85% of the maximum volume in each temperature zone. As we can see in the analysis above, the potential adjustment should be different for every zone. Theoretically the ambient and chilled fill-rates should be decreased as much as the average volume decrease and the frozen fill-rate should be increased as much as the average volume increase. It is advised not to do this.

Recently, Picnic has been using the new data without adjusting the fill-rate. I would not recommend to adjust the fill-rate based on this analysis, because it would increase the costs for ambient and chilled, while the practice now shows that the fuller totes are fine to work with.

Last but not least, this analysis is based on the idea that the current fill-rate of 85% works for Picnic. This analysis answers the question how to adjust the fill-rate such that in practice the totes will remain to be filled up to the same volume. However, one of the reasons that the artificial fill-rate is set to 85% has to do with the reliability of the data. Since the reliability has increased, the fill-rate should also be increased and not decreased.

Index