Moving the future of warehousing

Order processing in large-scale robotic mobile fulfillment systems

J. de Goffau

Faculty of Mechanical, Maritime and Materials Engineering

TU Delft Delft University of Technology

Challenge the future

Moving the future of warehousing

Order processing in large-scale robotic mobile fulfillment systems

by

J. de Goffau

in partial fulfillment of the requirements for the degree of

Master of Science
in Mechanical Engineering

at the Delft University of Technology,
to be defended publicly on Thursday January 10, 2019 at 9:00 AM.

Track:	Transport Engineering and Logistics	
Report number:	2018.TEL.8296	
Student number:	4324811	
Supervisors:	Dr. Ir. X. Jiang	TU Delft
	Prof. Dr. R. R. Negenborn	TU Delft
	R. L. Jordan	Eurobrain
Thesis committee:	Dr. J. A. Annema,	TU Delft

TU Delft Delft University of Technology

Abstract

Globalization is heaving its impact on many field. For the field of warehousing this development, in combination with the actual shortage of labor, means that many warehouse owners are forced to reduce their warehouse costs in order to stay competitive with warehouses in low-wage countries. On the other side the warehouse owners are pulled by recent technical innovations, which have reduced the drawbacks of automation. One of these recent innovations, is the mobile fulfillment unit-to-unit order picking system, which is presented in this paper. To make this system more affordable and hence to increase the competitiveness of warehouses in western Europe, research is required optimizing each of the system parts.

This research focuses on the order processing part of the system, which has changed considerably compared to existing mobile fulfillment order picking systems. In this system units are brought to two sides of a stationary order picker, who then transfers products from one unit to another unit. The order processing problem covers the sequencing of the unit visits at the order picking station, the batching of these units and the determination of the products that should be transferred.

Using insights from literature, several approaches are proposed to each of the problem parts. Each of these approaches is then tested and validated. Combining the approaches for each problem part leads to solutions addressing the full problem. All these solutions are tested against each other using a data set of one day of an existing warehouse. The tests are executed for multiple different order picking station configuration and for multiple data set sizes. This leads to an overview of the effectiveness of each of these solutions in terms of computational time and the reduced number of AGV visits.

From the results several conclusions can be drawn concerning the optimal solution for a specific warehouse. For an example warehouse in the Netherlands, a significant reduction of 25% can be obtained, regarding the number of visits, compared to the baseline approach. Which means the required number of AGVs can be lower, the investment costs for this system can be reduced and hence the competitiveness for this warehouse can be increased by implementing one of solutions presented in this paper.

Contents

1

Introduction

In today's fast-growing economy, many new companies are set up and existing companies are expanding in a rapid manner. More and more scarcity arises at the labor market and as a result people have the luxury to choose jobs they like. Less preferable jobs are refused and vacancies in companies offering these jobs cannot be filled. Warehouse managers, for example, have great difficulty finding people willing to do the physical intensive work they offer. At the same time companies are as focused as ever to cut the expenses on labor to stay competitive and profitable and hence they do not want to increase the wages for this kind of labor.
The combination of focusing on cutting labor costs and the scarcity of labor results in two options: automating the jobs or increasing the productivity of labor. The first option is only valid for the least complex jobs, in which humans can be replaced by machines or robots without too much effort. For the more complex jobs ways should be found to increase the productivity of the people doing the jobs.

Within the field of warehousing increasing productivity and cutting on labor costs have always been among the main goals, hence there has always been a reason for automation. But the recent increase of the lack of labor has made that reason even stronger, as companies are under threat of losing competitiveness with companies from other countries when their distribution process gets too expensive. The recent developments in technology that enable new opportunities for automation and productivity improvement, make the step to automation even more attractive and hence many warehouses are being automated nowadays.

In traditional warehousing, using the picker-to-parts systems, the costliest operation is the order picking operation and the costliest activity within the order picking operation is travelling. Reducing the costs for this activity is difficult to accomplish when sticking to the traditional order picking concept. To lower the costs, multiple alternative order picking concepts have been proposed and implemented reducing the amount of labor. But almost any of these systems has the drawback of requiring high investments and lack the scalability of the traditional order picking concept. To resolve these drawbacks, the robotic mobile fulfillment order picking system was introduced by Kiva, improving scalability and requiring less investment. The drawback of this system for most warehouses is that racks have to be developed, replenishment has to take place, orders need to be handled successively and at the end of

the line all boxes need to be collected.
Having inbound units with a turnover time of a day or at maximum a few days and an distribution process in which the outbound order sizes are comparable to the inbound order size, another concept can be implemented, which is called the unit-to-unit order picking approach, introduced by Eurobrain. In this concept units like pallets, racks or roller containers, are brought to an order picking station by AGVs. Where products are transferred from one unit to another. This concept seems to be very promising considering the total costs of the order picking operation and hence seems to be able to solve the lack of labor problem for several warehouses. This solution, however, can only be implemented when the required investment costs can be reduced to an acceptable level and the return on investment is high. Therefore, research is highly required to reduce the implementation costs of this concept as much as possible in order to contribute to the increasing competitiveness of companies dealing with expensive labor.

Research needs to be done to solving the several optimization problems that arise when implementing this order picking concept. This research focuses on solving one of them, which is the large scale order processing problem at the order picking station. This problem can be split in two parts; the linking of the inbound units to outbound units and the batching of these units at the order picking station. If this process is optimized, the number of AGV visits at the order picking station is reduced. That means that the number of AGVs required for daily operation can be lower, which leads to reduced investment costs. This optimization problem is NP-hard, as is proved by Boysen et al. [1], which means the number of steps required to solve it cannot be expressed as a polynomial function of the input, so computational time is a serious issue. Therefore, the goal of this research will be to answer the following question:

How can large-scale unit-to-unit based order processing problems be solved as effective as possible?

In which 'effective' means both a minimal number of unit visits to the order picking station and a minimized computational time. To answer this question, it first of all is required to review the state-of-the-art of the order processing problem in mobile fulfillment systems. Next the focus will be on what approaches can be applied to solve the linking part of the order processing problem and with what approaches the batching part of the problem can be solved. When the possible approaches are clear, the next point is how these can be combined into simulations each of which address both parts of the order processing problem. When that question is addressed and several simulations are generated the last point of interest is which of these simulations is the most effective one, both for a variable data set size and a variable order picking station layout. The sub-questions that will be answered in this research will be:

- What is the state-of-the-art regarding the order processing problem?
- What approaches can be applied to solve each of the problem parts individually?
- How can these approaches be transformed into simulations addressing the full problem?
- What is the effectiveness of the simulations in terms of the number of visits and computational time?

The report is structured as follows. In chapter 2 the problem outline will be described. Chapter 3 presents the state-of-the-art based on literature. Chapter 4 first off gives a further definition of the problem in the form of a mathematical model. Next it describes the followed methodology and the baseline for this research. Then it describes the possible approaches for the linking and the batching parts of the problem. And it finishes with the transformation from the approaches via algorithms into simulations and the validation of these simulation using a small-sized data set. Chapter 5 first presents the adaption and development that is required to increase effectiveness of the algorithms on large data-sets. Next, it describes the testing of the simulations using a real data-set of a distribution warehouse. It finishes presenting the results of these tests and discusses them. In chapter 6 conclusions are drawn and the main question will be answered using the answers on the sub-questions and an outlook will be presented on future research.

2

Problem Outline

The outline of the problem under research is given in this chapter. It starts with describing the main operations in traditional warehouses and the importance of an efficient order picking process. Next it indicates why optimizing the sub-activities of the order picking process has its limits within the traditional order picking concept. After that it presents several other available order picking concepts that improve efficiency, with their drawbacks. Then it presents the rack-moving order picking concept with its constraints. After which it presents the unit-to-unit based order picking concept and the importance of doing research to it. It finishes off by describing the problem in further detail and defining the project boundaries.

2.1. Warehouse operations

When dealing with warehouses generally two types of warehouses can be considered, the production warehouses and the distribution warehouses. A production warehouse stores materials and products for specific manufacturing or assembly processes. A distribution warehouse is defined by Hamberg & Verriet as a facility that stores products from many different suppliers for further distribution to their customers [2]. The latter is the type that this research is dealing with.
Within a warehouse many different kinds of activities are executed. These activities can roughly be categorized under four main operations [3]:

- The receiving operation, whose main task is to receive products and make them ready for storage,
- The storing operation, which deals with anything concerning storage,
- The order picking operation, whose main objective is ensuring that the right products, consolidated per order, are ready at the right time for shipment,
- The shipping operation, whose main objective is to load the right units in the right trucks.

Typically, a distribution warehouse has a storage with a huge variety of products. The customer's orders are typically small and do exist of a few lines with a few products per line. This often results in a complicated and cost intensive order picking process with a low productivity. In traditional warehouses

the order picking process is the costliest process with a share of 55 percent of all warehouse operational costs. [4] This means that increasing the efficiency of the order picking process has a significant impact on the total operational costs of the warehouse and hence is the main field for productivity improvement. [5][6]

2.2. Order picking efficiency

The order picking operation for a traditional warehouse is the activity by which the correct products in the correct quantity at the correct time are retrieved from the storage location in a warehouse as specified on a picking list, in order to fulfill customer orders. [7] To pick a product, an order picker normally has to travel to the picking location. When arrived he/she starts searching for the right product within that location and when found the order picker extracts the product from the storage. Often some other activities are involved as well, for example folding the cartons for boxes, building containers or doing some paperwork. Each of these activities cost a percentage of the total time required to pick a product, these percentages can be specified as follows [8]:

Activity	**Percentage of order picking time**
Travelling	55%
Searching	15%
Extracting	10%
Paperwork and other activities	20%

Considering that one of the key performance indices of warehouses is the total number of picks per hour, it can be concluded that the extracting activity is the only activity among these four that is productive, because the extracting of the product is what counts as an actual pick. In fact, travelling, searching and the paperwork and other activities are unproductive activities and the amount of time spent on them should therefore be reduced as much as possible. Which means that in the optimal case they would all be left out completely and the time it takes to extract a product would be minimized.

Reducing the time spent on extracting is often not possible for a standard process as it will always take a certain time to transfer a product from one location to another. Reducing the time spent on paperwork and other activities is often possible when replacing the paperwork by a digital system, such as a voice picking system or a button that should be clicked when a product is picked. Minimizing the time spent on searching can often be done by implementing a pick-to-light system or for instance an augmented reality solution.
Reducing the time spent on travelling is often a more difficult task, but at the same time it is also the most important one. Not only because traveling consumes the biggest heap of time, but also because the travelling is often a physically intensive activity, especially in warehouses where walking is the main form of transportation. In these warehouses it would be an idea to shift from walking to driving, but often that is not possible due to the warehouse layout or the organizational structure of the order picking process.

In literature two strategies can be found reducing the traveling time and distance that are exhaustive dealt with. The first one is the optimized routing strategy, the so-called picker routing problem. Choosing the shortest route to retrieve several products results in a shorter traveling distance. The other strategy is to optimize the storage assignment policy in such way that the products are stored at a smart location so that the average travelling distance to retrieve all products is reduced. This is called the storage location assignment problem.[9]

2.3. Improved order picking concepts

The implementation of the strategies and options mentioned above only lead to a certain reduction of the picking time per product. This is mainly caused by the fact that a full elimination of the travelling activity is not possible when maintaining the standard order picking concept. Which is the way it is organized in traditional warehouses, where an order picker moves to the products. This order picking concept is called the picker-to-parts order picking concept. In literature two other concepts are mentioned too [10]. The first one is the parts-to-picker concept, in which the picker is at a fixed location and the required products are moved to the picker. The second one is the automated picking concept, in which the human order picker is made redundant and order picking is fully automated. Examples of the parts-to-picker order picking concept are:

- Automated Storage and Retrieval Systems (ASRS'): Unit are retrieved from storage by storage and retrieval machine (SRM's) or by shuttles which are installed between aisles and present them to the order pickers in the sequence that they should be picked. A variation on this is the Vertical Lifting Machine (VLM).
- Carousels: Units with several products, such as racks, boxes or drawers are in a fixed sequence at a carousel and an order picker is standing at one stationary position. Then the units are presented one by one to the order picker, who picks the right products from them in the sequence that are required to fulfill the customers' order.
- Autostore shuttles: Units are piled up in a stack and shuttles ride over this stack to pick the unit of which one or more products are required for the next customers order and presents it to the order picker.

Examples of the fully automated order picking systems are:

- The robot order picker: The human picker in the foregoing concepts is replaced by a robot. Which makes them immediately fully automated.
- Automated Case Picking (ACP) or Modular Order Picking Systems (MOPS): Within these systems there is strictly speaking no actual order picking, but layer pickers and conveyors are used to present the products to a consolidator that put all products on a pallet or a container.
- Dispenser: units move over a conveyor and the right products are dispensed at the right time in or on the right unit to fulfill the customers' order.

All these systems have the big advantages compared to the traditional order picking concept that the traveling time is eliminated, which means that the productivity increases compared to the traditional warehouse and that the labor costs are reduced or even eliminated. One of the drawbacks of these

systems are that the investment costs to implement these systems are higher than for the conventional picker-to-parts process due to the systems and equipment that have to be installed. Another drawback is that scalability is very difficult. The system has to be at least of the size that is required to meet peak days demand. [1].

2.3.1. Rack-moving concept

In 2005 Kiva systems introduced a new parts-to-picker order picking concept, called the rack moving order picking concept, intended to avoid the drawbacks of the traditional order picking concept as well as these of the existing parts-to-picker concept [11].

In the rack moving order picking concept the order picker is at a stationary position and racks filled with products are brought to the order picker by automated guided vehicles (AGVs). These AGVs drive under the required racks in storage and lift them, move them to the order picker and when the order picker has taken the required products, the AGVs return the rack to the stack and head for the next rack. Because the order picker is stationary and the racks are presented successively, the productivity is higher and the labor costs are lower than in traditional warehouses. On the other hand, the scalability is maintained, because the fleet size can vary according to the number of products that needs to be picked and the investment costs are lower than these for the other parts-to-picker systems.

Initially there are several constraints to this rack-moving concept. The first one is the requirement that every inbound product needs to be placed in one of the specifically designed racks. The second one is that the orders need to be processed successively or at maximum a few orders at a time at each order picking station. The third one is that the orders are put in boxes that need to be collected at the end of the line for outbound transportation. And the forth is the need for replenishment to take place.

2.3.2. Unit-to-unit based order picking

Eurobrain has presented a generalization and expansion of the rack-moving order picking concept, called the unit-to-unit based order picking concept. This concept generalizes the rack-moving concept in such way that it is not restricted to racks only, but can move several kinds of units. The idea of the concept is that the inbound units from the suppliers are directly put in the stack and serve as racks. As in the rack moving concept, the units are moved from the stack to the order picker by AGVs, but the expansion is therein that the outbound units are also moved directly to the order picker by AGVs. This leads to an even further reduction in capital investment costs as there is no need for a system supplying and collecting the boxes at the end of the line, which means that the whole order picking system is dynamical, flexible and can be reorganized according to the targets for a specific day. Another advantage is that the orders do not need to be handled successively, but can be fulfilled in several steps, as it is possible to move the outbound units to a temporary stack after filling them partially and retrieve them later to finish them. Furthermore, with this concept there is no need to restack the inbound units to specially designed racks, as there is even no need for these racks at all.

The constraint of this concept is that the inbound units should have a turnover time of maximum a few days, otherwise it will result in a large inbound stack with many partially filled units, which would have a negative impact on the storage density.

As shown in the introduction there is an urge for companies to automate their warehouses, due to a lack

of employees and the need to reduce warehouse costs, in order to stay competitive. The unit-to-unit based order picking concept seems to solve this problem, for a specific group of warehouses, in a better way than other existing order picking concepts. However, this solution can only be implemented when the investment costs required can be reduced to an acceptable level. Therefore, research is highly required to make this concept cheap enough to be implemented in the field of warehousing and give companies the possibility of competing against other (foreign) warehouses having lower labor costs in their distribution process.

2.4. Problem description

When implementing the unit-to-unit order picking concept in an existing warehouse the aim is to reduce the personnel costs as much as possible while keeping the investment costs low and the total productivity at the same level. This means that the amount of order pickers and the amount of AGVs used should be as low as possible while maintaining the original warehouse throughput. Reducing the total number of AGVs in a warehouse means that the productivity per individual AGV should be maximized. To do so, the following optimization problems need to be solved as optimally as possible:

- The vehicle dispatching problem: This optimization problem deals with the question about which assignments should be dispatched to which AGV in order to have the least idle riding time and thus the highest productivity per AGV.
- The storage assignment problem: This problem treats the positioning of each unit in the storage in such way that the total average driving distance from the storage to the order picking station and other locations is minimized.
- The vehicle routing problem: This is an optimization problem that is about the question what the optimal set of routes for a fleet of vehicles is to travel in order to execute all assignments and have the lowest possible total travelling time.
- Mobile robot-based order processing problem: This is a less well-known problem as it deals with the sequencing of units at the order picking station. This sequencing needs to be done in such way that the total number of visits of units at the order picking station is as low as possible, so that the number of AGVs can be minimized.

The last problem of these four is the problem that this research deals with. This mobile robot-based order processing problem needs to be solved for an minimum number of unit visits at the order picking station. Every time a unit is to visit the order picking station, the AGV first has to drive to the storage location, lift the right unit, drive to the order picking station, wait its turn, drive to the correct place at the order picking station, wait until the products are transferred, drive back to storage and drop the unit at the right location. This means the number of unit visits is very much related to the number of AGVs required in the system.

The order processing problem came into existence at the moment that the rack-moving order picking concept was introduced. With the expansion of this concept to the unit-to-unit based order picking concept, the order processing problem also expanded in complexity. In the rack-moving concept the problem was restricted to finding the optimal combination between a unit with many products and a

number of small orders. With the new concept the optimal combination has to be found between two units full of products. In the rack-moving concept the orders should be handled successively, but in the new concept this restriction does not exist anymore. The increasing complexity of the picking concept also added an extra dimension to the problem, as it enabled to let one or several units at an arbitrary side of the order picking station stay in its place and let other units at the other side of the picking station pass by. The next moment the unit at the other side of the picking station can be kept in place and the units at this side can be changed. In other words, there is no restriction to the sequence in which the units are filled.

This extended version of the mobile robot-based order processing problem is the problem that this research will deal with. It will focus on solving this problem in such way that the number of unit visits to the order picking station is minimized. This is why the main question was stated as follows: "How can large-scale unit-to-unit based order processing problems be solved as effective as possible?"

2.5. Project boundaries

To increase clearness of the research it is important to eliminate model complexities that are not required to answer the research question. When minimizing the number of AGV visits at the order picking station, some factors can be left out of consideration because they have little or no impact on the outcome. The most important one that can be left out is time. Finding the optimal number of visits of units at the order picking station can be done without taking time into account. The required number of visits does determine the number of AGVs required and the time it takes to complete all jobs, but the number of AGVs and the time does not have much influence on the number of visits.
Leaving time out of consideration means that expedition time, the productivity per order picking station, the number of AGVs and the starting time of the order pickers do not need to be taken into account. This means adaptions have to be made to the outcome of this research in order to implement it in warehouses in which, for instance, expedition time is of crucial importance.

The second thing that is left out of consideration for this research is the number of order picking stations. Handling one unit at two order picking stations successively, will reduce the time one visit takes. But the performance of the system and the reduction of time, depends heavily on the layout of a warehouse. Considering that this optimization can be applied after solving the order processing problem, it means that the order processing problem can be solved first for more general warehouses. And that the optimization of splitting the orders over several order picking stations can be applied afterwards, customized to the specific warehouse layout.

3

Literature

This chapter presents what has been done so far in the fields related to the research problem, which approaches were taken, and which solutions were proposed. The literature study is divided into four parts. The first part focuses on research that is done to the same problem and to research done for other problems in the same field. The second part focuses on research done to comparable problems in other order picking systems to figure out if applied algorithms in these field can be deployed to solve the problem in this research too. The third part focuses on related fields, such as container terminals, to identify if comparable problems are addressed there and if they are, what approaches were followed and whether these solutions can applied in the context of this research. The fourth part focuses purely on research done to algorithms that possibly can be used when developing algorithms for the problem addressed in this research. These four parts of the literature study indicate the structure of this chapter. The chapter finishes discussing the obtained results and the insights gained for this research.

3.1. Robotic mobile fulfillment systems

The general name in literature for the rack-moving order picking concept and the unit-to-unit order picking concept is 'robotic mobile fulfillment system'. The first part of this literature study focuses on getting an insight in what is described already in literature regarding robotic mobile fulfillment systems. And to figure out whether the extended order processing problem, which is addressed in this research, is identified already in literature too. This study starts with referencing the first paper mentioning the robotic mobile fulfillment system and continues with referencing each paper up to now addressing both the robotic mobile fulfillment system and the order processing problem.

The first one in literature presenting the basis of the robotic mobile fulfillment system order picking concept was the founder of Kiva System LLC with patenting parts of the system [11]. The first permanent implementation of a robotic mobile fulfillment system was in 2006. In 2007 the three founders of Kiva presented a paper [12] in which they described the Kiva system in certain detail and clarified why this system has more advantages and less drawbacks than the existing systems. The optimization, computational and organizational problems of implementing such a system are mentioned, but not into full detail.

What they do mention is that jobs are first of all assigned to a certain order picking station. This is done using heuristics comparing each job with the jobs already assigned to the order picking stations, to get the highest similarity and thus to maximize the number of items that can be picked from one unit. Once a job is assigned to a certain station, the units that can bring the products for that order are compared with heuristics combining minimizing the distance to the order picking station and maximizing the number of products on each unit. One of the concluding messages of the paper is that the founders of Kiva Systems expect the Kiva approach to spread through the industry, and when it does, many interesting new computational and organizational problems need to be solved.

In 2008 one additional paper was written by two of the co-founders describing the future challenges of the implementation of the Kiva system and again the allocation of the units to the order picking station is mentioned as one of the challenges. To encourage research to the challenges mentioned in the paper, a simulation environment (Alphabet Soup) is launched in which job assignment and the AGV assignment algorithms for the Kiva system can be tested. This simulation environment works outs to be still available. [13]

Furthermore in 2008, an article was written in which the start of the Kiva Systems company was explained, providing some more details on how their assignment systems work. According to the writer, optimizing the assignment of racks to the station and organizing the inventory is really difficult to do from a centralized computer as it is a NP-hard problem. Kiva Systems solved it by introducing a multi-agent system running on the centralized computer, the AGVs and on the computers at the picking station, optimizing each their own task. Besides that, they adopted heuristic methods, like Greedy algorithms, that produce sub optimal solutions to complex problems. [14]

In 2011 a paper was published giving more insight detail in what allocation problems come into existence when implementing the Kiva order picking concept [15]. Enright & Wurman describe the mobile fulfillment systems, as a relatively new and understudied class of resource allocation problems. The mobile fulfillment systems bring together the computational problems for many different fields, such as scheduling, data mining, decision making under uncertainty, learning and classic optimization.

According to them the main objective for optimization is a dual objective function. At one hand keeping the order pickers as busy as possible while on the other hand minimizing the amount of equipment necessary, particularly AGVs. Keeping the pickers fully occupied results in minimal operational expenses and minimizing the necessary equipment results in minimal capital expenses.
As mentioned before, the Kiva concept starts with assigning an order to a station. This is called *the order allocation problem*. After that a unit is selected that should be brought to the station, called *the inventory unit selection problem* and then an AGV should be assigned to carry the unit to the order picking station, called *the robot allocation problem*. In the paper is mentioned that not very much can be done to the driving length once an AGV is assigned to a unit. The biggest lever for reducing the number of robots needed, according to them, is increasing the average number of items picked from each unit that arrives at the order picking station. This concept is then referred to as 'pile-on' and four different types of pile-on are distinguished:

- Bin pile-on: In which multiple products from one bin on one unit can be transferred into multiple different orders at the same time at one order picking station.

- Face pile-on: In which multiple products can be picked from one face of the unit at the same time at one order picking station.
- Station pile-on: The total number of products that can be picked from one unit at one order picking station. The unit can rotate in the meantime.
- Mission pile-on: The total number of products picked from one unit in one mission at possibly multiple order picking station.

They mention that the Kiva system is very attractive from an algorithmic point of view, due to the fact that the overall complex problem can easily be cut into small subproblems, which then can be solved rather easily. It is to be seen they write whether the optimizations that address the global problem outperform the optimizations that address the decomposed optimizations.

According to them the inventory unit selection problem corresponds at a certain moment to the multi-set multi-cover problem, but then with an extra dimension that rarely is explored. When one of the orders at the station is finished another order will take its place, in the unit selection problem this part could be taken into account to maximize the station pile-on. When proposing a solution, they say, it should be considered whether it is a good idea to select units that are already selected by other stations.
For the order allocation problem, which is mentioned as the most critical problem in the system, it is clear, they say, that combining orders with many similar products is going to improve the pile-on. But the question is whether it is going to be possible to anticipate on the distance to the inventory units and on the other units that are present at the station at that time. Making good decisions at this point could also increase pile-on and reduce the time spent taking units to the station. Another point that requires attention is the prioritization of orders that are almost due, to make sure they will not get overdue.

When a good working version of the system is implemented, there still is much space for optimization; Kiva systems managed to realize a ten percent throughput improvement in one year and a half [16]. Knowing that these gains are possible it is clear that research to the presented problems is useful. Having the papers mentioned above as a basis defining a set of clear problem definitions, several papers were written addressing one or more of these mobile fulfillment system problems. Such as the inventory allocation problem [17] [18], the battery swapping problem [19], the layout problem [20], [18], and the path planning problem [21], [22].
Xiang et al. [23], Merschforman et al.[24] and Boysen et al. [1] all propose a solution for the sequencing of the orders and/or units in a mobile fulfillment system which will be discussed in more detail below.

Xiang, Lui & Miao [23] propose batching the orders as a solution to the optimal sequencing of the orders. According to them batching the orders in such way that the similarity within a batch is as high as possible, will gradually reduce the number of unit visits at the order picking station. The order batching problem is NP-hard, so computational time becomes a very important issue when the number of orders increases. Due to this computational time, exact algorithms solving this kind of NP-hard problems can only be used on very small-scale problems. The authors therefore developed an algorithm to reduce the computational time and get an approximate solution. The algorithm exists of two steps:

- An initial solution is generated by maximizing the number of combinations of products in two orders, that are also available on one unit or by minimizing the number of unrelated combinations at the units. Highly associated orders are linked this way and low associated orders are separated.
- The Variable Neighborhood Search (VNS) is applied to optimize the initial solution.

After using this algorithm to determine the batches in which the order need to be picked, the next step they executed was applying a Greedy strategy to find the best unit combinations for each batch. The time required to solve this problem is within seconds for medium sized problems, but the computational time grows exponentially with increasing complexity. The question therefore is whether this approach is usable for the warehouse size addressed in this research. It concludes that optimized storage assignment and optimized order batching can noticeably reduce the number of visits of units.

Merschformann et al. [24] propose several different solutions for order sequencing and for unit sequencing. Starting with the order sequencing problem they assume that all picking station are fully filled with orders. When one order is finished a new order has to be chosen from the backlog and assigned to that picking station to take its place. This way no order batching is done, but the decisions are made when an order leaves the station. To make this decision several rules can be applied:

- Random: the new order is chosen randomly from the backlog
- FCFS: The orders are handled in the same way as they appear in the backlog.
- Fast-lane: most orders are chosen randomly from the backlog, but one slot at each order picking station is kept open to construct a fast lane. This fast lane can be filled with orders for whom all lines and all products are available on the next unit, which means they can be finished immediately when the next unit appears.
- Common-Lines: The order that has the most lines in common with the orders that are already assigned to the picking station is chosen from the backlog.
- Pod-Match: The order that matches best with the units (called pods in Kiva jargon) heading to the station is chosen from the backlog.

All these rules are tested in a simulation that approaches the real-world application of the Kiva system, called RAWSim-O [25]. This simulation is open source and has implemented several algorithms for many problems, such as path planning, unit selection and order assignment. Testing the proposed solution in this simulation showed that the Pod-Match rule is the rule that results in the highest performance of the system, i.e. the maximum throughput of orders within 48 hours. Once orders are assigned to an order picking station, the best unit has to be chosen from the storage to fulfill these orders. The best unit can never be one without any product for the orders that need to be fulfilled, so these units are always eliminated from the considerations. To choose this best unit six different rules are proposed:

- Random: A unit that offers at least one useful product is chosen randomly from storage.
- Nearest: The unit with the shortest travel time to the order picking station with at least one useful product is selected.

- Pile-on: This rule selects the unit that offers most useful products to fulfill the order at the station from the storage.
- Demand: The unit whose content is most demanded by the backlog orders is chosen from storage.
- Lateness: Units are selected focusing to fulfill the orders whose due time is most imminent.
- Age: Units are selected with a focus on fulfilling the orders that spent the longest time at the picking station.

Testing these rules in the RAWSim-O simulation for many different situations showed that the Nearest rule on average performs best, closely followed by the Age rule and the Pile-on rule. One of the conclusions that resulted from that study is that varying the decision rule for solving the Order Assignment Problem affected to throughput rate the most. The best rule resulted in a double throughput rate compared to the worst rule. Therefore, their advice is that warehouse operators should pay most attention to the Order Assignment decision problem.

Boysen et al. [1] propose three different heuristic approaches to optimize the combination of order sequencing and rack sequencing for one picking station. The first approach is determining the rack sequence for a given order sequence. The order sequence is generated randomly, and the rack sequence is optimized for this specific order sequence. The second approach is determining the order sequence for a given rack sequence. And the third approach is alternatingly solving the first and the second problem. All three approaches get to a solution using a dynamic programming model implemented in a Simulated Annealing algorithm. The algorithm proposed in the third approach obtains a solution in quite less time than the benchmark algorithms which is a MIP model implemented in CPLEX while it is able to approach the same optimality. But, as the computational time increases exponentially on increasing complexity, this algorithm is not useful for the warehouse size addressed in this paper as the computational time should be expressed in years then, instead of seconds.

In table 3.1 an overview can be found showing for each referenced paper which algorithm was proposed to solve both the order assignment problem and the unit assignment problem.

3.2. Comparable order picking systems

Within other order picking systems many optimization problems exist as well. The two problems that are most related to the mobile robot-based order processing problem are the order batching problem in the picker-to-parts warehouses and the ASRS order batching optimization. The order batching problem in picker-to-parts warehouses are almost always based on the successive travel distances between the picking locations and therefore alter from the mobile robot-based order processing problem in which the pickers stay at their positions and products are brought to the picker.
The order batching in parts-to-picker systems, such as the ASRS, VLM and carousel are, however, very similar to what can be used in the mobile robot-based order processing problems. Multiple papers are written on this subject for specific systems. Hwang, H. et al. [26] published a paper on order batching in an ASRS proposing several algorithms for order batching that focus on the similarity between the orders. Elsayed E. et al. [27] proposed an algorithm batching the orders based on the tardiness and earliness of the orders. More recently Nicolas, L. et al. [28] proposed a Mix Integer Linear Program solved with CPLEX which was able to reduce to overall picking time with approximately 30% but the computation

Paper	Order assignment	Unit Assignment	Applicable to large problems?
Wurman, D'Andrea & Mountz [12]	Heuristically based on earlier assigned orders	Heuristically based on distance and number of similar products	Yes
Guizzo [14]	Multi-agent local heuristic solvers optimizing own task	Multi-agent local heuristic solvers optimizing own task	Yes
Enright & Wurman [15]	Assign units, with several orders, based on: inventory required, open lines, products on available units	Multi-set multi-cover problem with extra dimension	Yes
Xiang, Lui & Miao [23]	Order batching: association/alienation + VN	Greedy strategy	No
Merschformann et al. [24]	Assigning when vacancy at station in 5 rules: Random, FCFS, Fast-lane, Common-Lines, Pod-Match	No double assignment and 6 rules: Random, Nearest, Pile-on, Demand, Lateness, Age	Yes
Boysen et al. [1]	Three options: • Optimize heuristically for fixed initial unit order • Fixed initial order • Optimize alternatingly for optimized unit order or keep fixed	Three options: • Fixed initial order • Optimize heuristically for fixed initial order sequence • Optimize alternatingly for optimized order sequence or keep fixed	No

Table 3.1: Overview of referenced papers solving the Order Assignment problem and/or the Unit Assignment problem

Paper	Order assignment	Applicable to large problems?
Hwang, H. et al. [26]	Order batching based on similarity	Yes
Elsayed E. et al. [27]	Order batching based on tardiness	Yes
Nicolas, L. et al. [28]	MILP in CPLEX	No
Nicolas, L. et al. [29]	Simulated Annealing	No

Table 3.2: Proposed solutions by referenced papers to the Order Assignment problem

was stopped after thirty minutes as half an hour was regarded as a long time. After searching for a solution for this while presuming the optimality of the result, Nicolas, L et al. [29] proposed a simulated annealing approach to solve this problem within minutes while obtaining comparable optimality. These real-world cases both existed of only a few hundred orders and computational time has already to be expressed in minutes, which means that scaling it to thousands of orders will result in an undesirable long computation time.
Table 3.2 gives an overview of the mentioned papers with their proposed solution to the order assignment problem.

3.3. Other fields with related problems

The field of warehousing is in some respects very much related to the 'warehousing' done at container terminals. Storing and retrieving to and from stack can very well be compared to storing and retrieving from stack. This field, however, cannot be used to solve the order processing problem as no order picking and order batching is done at container terminals. Searching for solutions within other fields than warehousing did not deliver many new insights. There do exist many optimization problems and techniques in other fields, but none of the cases considered is comparable with the problem addressed in this paper, except the optimization for signalized junctions. In that field some other batching is applied too and sequences have to be determined. Improta, G. & Cantarella, G. [30] proposed an algorithm for solving this by translating the problem into an Binary Mixed Integer Linear Programming problem and solved it with a Branch and Bound method using Simplex. This approach is only valid for small scale problems where computational time does not matter too much. A multi-product transportation problem with costs per link between supplier and customer instead of costs per product could be a comparable problem. But no algorithms have been found so far addressing that problem.

3.4. Useful algorithms

In the field of operations research many algorithms exist that focus on solving optimization problems. Examples of standard problems are: The Transportation Problem, the Assignment problem and the Travelling Salesman Problem. And examples of algorithms to solve them are: The Simplex Method, the Dual Problem, Integer Linear Programming, Greedy Heuristics, Dynamic Programming and Simulated Annealing [31]. All these problems are NP-hard or NP-complete and thus is finding the exact solution for huge problems not possible within polynomial time. Which means a heuristic approach has to be applied to get a near to optimal solution. Many papers do propose algorithms for solving a certain kind of problem heuristically, but it is not always clear whether the proposed algorithms can be applied at large scale without running into computational problems and whether they will perform as

good as proposed under different circumstances. One of the things that becomes clear from most of the papers is, which is in line with the expectations, that the computational time increases when the problems size increases. The drawback of many heuristics algorithms is that the computational time increases exponentially, which means that very large problems will take too long to be usable.

3.5. Discussion

Due to the fact that robotic mobile fulfillment systems have taken off quite recently and the amount of papers addressing it is still relatively small, it was possible in this literature review to get a near to complete overview of state-of-the-art of this field. From the papers involved several insights could be deducted which are discussed below.

The first insight that is gained from these papers is that solving the order processing problem cannot be done using algorithms finding exact solutions, because they do require an unreasonable amount of computational time for very large problems. This means that algorithms and algorithm fields such as the Simplex method, the Linear Programming field and Dynamic Programming field's algorithms cannot be used. As an alternative, meta-heuristic algorithms are used in many papers. Implementing these algorithms means that the optimal solution will not be obtained, but the result will be an approximation of it.

The second insight from these papers is that there exist two main levers to reduce the number of unit visits at the order picking station. The first one is maximizing the number of products picked for each order, from each visiting unit. This is called 'pile-on'. The amount of pile-on is inversely proportional to the number of unit visits, which means maximizing pile-on results in minimizing the number of unit visits. The second lever to reduce the number of visits is batching the orders at the order picking station, so that products for multiple orders can be picked at the same time. This order batching can be done in two ways. One way is to group multiple orders into batches and finish one batch before the next one is handled. Another way is to define a starting batch and assign a new order every time an order within the batch is finished. The latter means that choices can be made based on the most actual warehouse situation, for instance taking it into account when some products arrive overdue to the warehouse, or when some orders require prioritization due to their expedition time.
The papers on alternative order picking systems, that were researched to figure out if approaches presented in them could be used in this field, show that order batching can surely increase the efficiency of the total order picking process. Furthermore, that order batching is applied in related picking systems too and that for large problems, multiple heuristic algorithms are proposed. They, however, are not applied to very large problems and their computational time is often quite significant, which is not very promising for applying them to very large problems.
Within the other fields no useful approaches where obtained. The study to useful algorithms filled the algorithm toolbox with standard algorithms and problem approaches that can be used in the algorithm generation process.

4

Method and simple case validation

As described in the previous chapter, the literature study showed that new approaches using meta-heuristic algorithms are demanded to solve the large-scale order processing problem in the unit-to-unit order picking system. Furthermore, it showed that the order processing problem is often split into two parts: order batching and unit assignment. In the unit-to-unit order processing problem, the 'orders' of the standard order processing problem are replaced with units. This means that 'order batching' is changed to 'unit batching', which is being applied to both side of the picking station. This also means that the '(unit to) order assignment' is changed to 'unit to unit' assignment. Which for the sake of clearness will be called 'unit linking' henceforth. In this research the unit linking is called the first part of the unit-to-unit order processing problem and the unit batching is called the second part.

The first section of this chapter starts with describing the method to find and validate these approaches. Then a further definition of the problem is given using a mathematical model and it finishes with a definition of the baseline of this research. The second section presents what new approaches are generated to solve the first part of the order processing problem, which is the linking of in- and outbound units. The third section presents the new approaches for the batching part of the order processing problem which is the second part of the problem and the filtering process to filter out the best approaches. The fourth section describes how the found approaches are combined into solutions addressing the full unit-to-unit order processing problem. Furthermore, this fourth section describes how these solutions are transformed into simulations, which enables testing the effectiveness of the solutions. The fourth section finishes with a description of the validation and verification process of the modules and the simulations.

4.1. Method

In the next part of this research, which focuses on finding these new approaches to the unit-to-unit order processing problem and testing these, the following methodology will be applied. First of all for both the linking part and the batching part, several approaches will be generated that are expected to be able to deal with larger data sets. These approaches will be transformed into modular algorithms, each of which will at first be verified and validated using a small data set. Then the effectiveness of each of these algorithms will be tested on a specific case, which is a real data set of one day of a

distribution warehouse. Information on the performance under different conditions will be gained from tests with multiple data sets and multiple order picking station configurations. The gained results will be discussed and conclusions will be drawn from them.

4.1.1. Mathematical model

To fully define the order processing problem for the unit-to-unit order picking concept a mathematical model will be presented, which is partly based on the model presented by Boysen et al. [1]. Within this model several assumptions are made:

- Only one order picking station is present in the system. This means there are no interdependencies with other order picking stations, i.e. no units that need to be available in two time slots at once. When multiple order pickings stations would be used, the output can be redefined by implementing a blocking mechanism that blocks units for several time slots when they are used once. Assuming that the number of units is large, the blocking of a few of them will not influence the performance of the system too much and thus does the model with this simplification still reflect reality.
- Picking lines comprising several products are split up into lines with only one product. This means that one original picking line can be picked from multiple units. Which is a common practice in warehousing and thus does reflect reality.
- The products are spread over several units, which means that often one product can be picked from several inbound units. This is an assumption based on reality; most often products are spread across different units and thus it does reflect reality.

The order processing problem as described above can formally be described as follows. The required output consists of two lists: the first one is list with the unit sequence, denoted as d, of inbound units i and outbound units j divided into time slots t. The second one is a list of picking lines p specifying for each t which products s should be picking in that slot. Notice that the number of time slots can vary but can never be higher than the number of products that should be picked as it is required that every time slot at least one product should be picked. d is only feasible when each order is scheduled exactly once and p is only valid when all products required by the outbound unit in a certain time slot is provided by the inbound units available in that time slot.

Definitions:

I = Set of all inbound units

J = Set of all outbound units

S = Set of all products

T = Set of all timeslots

Indices:

i = Index of inbound unit $i \in I$

j = Index of outbound unit $j \in J$

s = Index of product $s \in S$

t = Index of slot $t \in T$

Decision variables:

$U_{s,j,t}$ = Binary variable: 1, if product s is transferred to outbound unit j at slot t

$V_{i,t}$ = Continuous variable: 1, if inbound unit i is visiting at slot t

$W_{j,t}$ = Continuous variable: 1, if outbound unit j is visiting at slot t

$F_{j,t}$ = Continuous variable: 1, if outbound unit j is filled at slot t

$A_{i,t}$ = Continuous variable: 1, if inbound units visiting at $t-1$ and t differ

$B_{j,t}$ = Continuous variable: 1, if outbound units visiting at $t-1$ and t differ

Parameters:

C_i = Capacity at inbound side of the order picking station

C_j = Capacity at outbound side of the order picking station

$Y_{s,j}$ = Set of products s at outbound unit j

Z_s = Set of inbound units providing s

Objective:

$$Minimize: F = \sum_{t=2}^{T} A_{i,t} + \sum_{t=2}^{T} B_{j,t} \tag{4.1}$$

Subject to:

$$\sum_{t=1}^{I} V_{i,t} \leq C_i \qquad \forall t = 1, \ldots, T \tag{4.2}$$

$$\sum_{j=1}^{J} W_{j,t} \leq C_j \qquad \forall t = 1, \ldots, T \tag{4.3}$$

$$\sum_{t=1}^{T} U_{s,j,t} \geq C_j \qquad \forall j = 1, \ldots, J, \qquad s \in Y_{s,j} \tag{4.4}$$

$$2U_{s,j,t} \leq F_{j,t} + \sum_{i \in Z_s} V_{i,t} \qquad \forall j = 1,...,J, \qquad \forall s \in Y_{s,j}, \qquad \forall t = 1,...,T \tag{4.5}$$

$$A_{i,t} \geq V_{i,t} - V_{i,t-1} \qquad \forall t = 2,...,T, \qquad \forall i = 1,...I \tag{4.6}$$

$$B_{j,t} \geq W_{j,t} - W_{j,t-1} \qquad \forall t = 2,...,T, \qquad \forall j = 1,...J \tag{4.7}$$

$$U_{s,j,t} \in \{0,1\} \qquad \forall s = 1,...,S, \qquad \forall i = 1,...I, \qquad \forall t = 1,...,T \tag{4.8}$$

$$1 \geq F_{j,t} \geq 0 \qquad \forall j = 1,...,J, \qquad \forall t = 2,...,T \tag{4.9}$$

$$A_{i,t} \geq 0, \qquad \forall t = 2,...,T \tag{4.10}$$

$$B_{j,t} \geq 0, \qquad \forall t = 2,...,T \tag{4.11}$$

Objective 4.1 minimizes the number of unit changes at the order picking station and thus the number of unit visits. Constraint 4.2 and 4.3 assure that no more in and outbound units are assigned to the order picking station then is allowed. 4.4 states that each product required to fulfill the outbound units is picked. Picking is only allowed when both the right inbound and the right outbound units are available due to 4.5. 4.6 and 4.7 track the amount of unit changes at the order picking station. 4.8 forces $U_{s,j,t}$ to be either zero or one. 4.9 restricts $F_{j,t}$ to be between zero and one as each order cannot be filled more then completely. 4.10 and 4.11 restrict the rack changes to be not lower than zero. [1]

4.1.2. Baseline

Due to the fact that there is currently no warehouse around in which the unit-to-unit order picking concept is implemented, defining an existing situation as a baseline is not possible. Therefore, another approach has to be followed. In this research the baseline for solving the unit-to-unit order processing problem, will therefore be defined as the assembly of the most straightforward approach to solving each individual problem part.

For the first part this means that the combination of inbound and outbound units is found in the following manner. First a random inbound unit is taken. For each product at that unit an outbound unit is taken randomly requiring that product, and the product is transferred. This is done until the inbound unit is empty. Then the next inbound unit is taken, until all inbound units are empty.

The second part of the problem, which is the batching process is split into two halves. The first one, which is the creation of the batches is also solved taking the most straightforward approach. Which is assuming that the capacity of the order picking station is only one unit at each side. The result is that each unit is an individual batch and hence no algorithms have to be applied to find batches.
The batch sequencing, which is the second half of the second part of the problem, is done randomly, which means that the first batch found, will be the first batch handled. When it appears that one of the outbound units that was at the order picking station already for the previous batch can be used in this batch again, then this also results in a reduction of one visit.
This baseline will be implemented as a simulation to test the performance of other algorithms against this one. It could be an option also to take the first outbound unit as a starting point and choosing the inbound units randomly, but this approach is chosen randomly too, without having an idea on which approach would be the best.

To clarify the baseline and the mathematical model an example scenario is made up, which can be found in B.

4.2. Unit linking

This section describes the development of new approaches that will be able to solve the first part of the unit-to-unit order processing problem, which is the linking of inbound and outbound units. An inbound and an outbound unit are denoted as linked when products should be picked from that inbound unit and transferred to that specific outbound unit.

4.2.1. Lessons from literature

The literature study showed that solving the linking part of the order processing problem optimally would require too much computational time for the problem size addressed in this research and hence is not a valid option. Multiple tests executed in the first stage of this research, with for instance CPLEX optimization studio or a Hungarian algorithm solving a combination of the allocation problem and the travelling salesman problem, confirmed this lesson from literature.
As an alternative for that, in literature heuristic approaches were proposed that are more promising considering computational time than approaches finding exact solutions. Taking this into account for this research, finding the exact solutions will not be the goal of the approaches that will be proposed in this chapter. Another lesson from literature was that maximizing pile-on is one of the biggest levers to reduce the number of unit visits at the order picking station. Taking these results of the literature study into account and reconsidering what could be useful to reduce the number of unit visits in the linking process, lead to the following approaches.

4.2.2. Maximizing Pile-On

The first approach is based on maximizing pile-on (MPO) at the order picking stations. This means that the average number of products picked at the order pickings station per unit visit should be maximized for all unit visits. Finding the exact best solution for this approach, using the brute force method, would require calculating all options and taking the best results of all options. This, however, would be computationally very expensive as the number of options grows very fast with the number of units.

If the number of inbound units would be I and the number of outbound units would be J then the number of options that should be calculated before finding the best option would be:

$$(I * J)! \tag{4.12}$$

Doing this for the targeted problem size, this would lead to a number of options that would exceed $10^{50,000,000}$. This number of options is clearly impossible to solve within reasonable time on a normal computer. And this is without taking the products that should be transferred into account, which could blow up the number of options even more. This is in line with what literature showed, namely that finding an exact solution is not an option.

Applying heuristics can gradually reduce the number of computational steps. One of the heuristics that is applicable and is very lightweight regarding required computational performance, is the Greedy search algorithm. This is an algorithmic strategy that focuses on finding the next best option at every stage, targeting to approach the global optimum. Due to this approach the risk of not reaching the global optimum, but only a local optimum is tremendous. This risk could be lowered a bit by adding the Genetic Algorithm or other alike algorithms. These algorithms focus on improving the change to find the global optimum, but at the other side they increase the number of computational steps required. Therefore these extensional algorithms will be left out in this research, but can be included in further research.
Applying the Greedy search approach to the MPO approach can be done by first calculating the amount of products that can be transferred for each of the combinations of in- and outbound units. Performing this operation on the example problem as defined in Appendix B and shown in 4.1, results in the matrix shown in table 4.1. From this matrix the highest amount of products will be the first link that is chosen. Then the products are removed from the units in the data set, the matrix is updated and the next best amount of products is calculated, until all in- and outbound units are empty.

	AA	BB	CC	DD
A	3	1	2	1
B	1	1	1	2
C	2	1	2	3
D	2	1	2	1
E	1	2	1	2

Table 4.1: Amount of transferable products

	AA	BB	CC	DD
A	3	1	2	1
B	1	1	1	2
C	2	1	2	3
D	2	1	2	1
E	1	1	1	2

Table 4.2: Number of order lines

4.2.3. Maximizing Order Line pile-on

The second approach that is generated is closely related to the maximizing pile-on approach. Maximizing pile-on means maximizing the actual number of products picked in a warehouse. In many warehouses products are often not ordered separately, but together with several other products of the

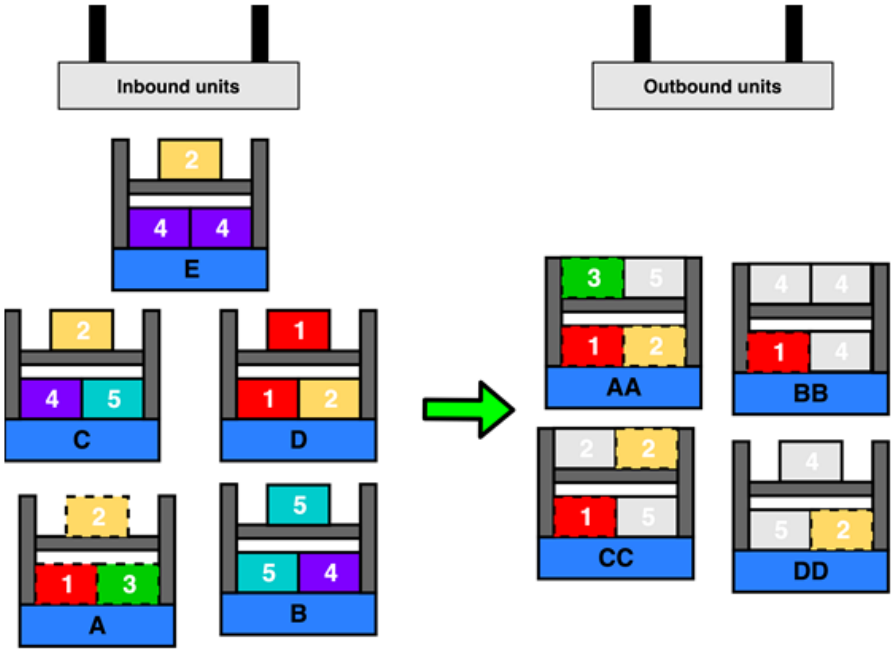

Figure 4.1: Example problem for MPO

same and a bunch of products of other kinds. The total bunch of products is called an order and a number of products of an individual kind is called an order line. When locally focusing on maximizing the number of products, like the first approach does, it can occur that multiple order lines are picked half. Which means that all remaining products need to be transferred another time by different units. In the worst case this means that units are brought for the remaining products, probably could have provided all products that initially where required and the first visit would be redundant. To find out whether focusing on the amount of products is the best option, or that it is probably better to focus on transferring whole order lines at a time, the second approach focuses on maximizing the number of order lines (MOL) that can be transferred at once.

To find the maximum number of order lines transferred at a time, the amount of order lines that can be transferred is listed and the combination of units between which most lines can be transferred is chosen as the first link. For the example problem presented in appendix B, the list of number of order lines is shown in table 4.2. The Greedy Search algorithm is implemented exactly in the same way as is done for the first approach. Every time the products are picked and the number of possible order lines that can be transferred is recalculated. This process repeats itself until the whole data set of provided products on the inbound units and the required units at the outbound units is empty.

4.2.4. Product Based Optimizing

Taking the first best option globally, as the first two examples do, can result in making bad decisions locally. An example of this can easily show this. For instance when considering two inbound units and two outbound units providing or requiring the same kind of product. If the first inbound and the second outbound unit supply an amount of two and the second inbound and the first outbound unit an amount of one of that product. This product can now be picking in two ways: The products on the first inbound unit can be transferred to the first outbound unit. This means that three links are at least

required to transfer all products. The second way is to transfer the products from the first inbound unit to the second outbound unit, which results in only two links. Although both the MPO and MOL approaches would choose the right combination when only these products would be on the units, this local imperfection can occur on units full of products.

This example shows that global optimization can lead to unnecessary links. Local optimization at the other side can lead to sub-optimal performance globally. Therefore, it is necessary to weigh up carefully if a local optimization approach would yield better results than the first two approaches. This will be done with the third approach, the product based optimizing approach (PBO). This approach will focus on minimizing the number of links per individual product type. This minimization will be done using a custom made algorithm, which will find the best links to transfer all products of that certain type. When for all types the corresponding links are found, the next step in the approach will be to combine all these links and remove the duplicates. All links together form a valid approach to the order processing problem in warehouses.

The custom made algorithm that will be used in this approach is created using the following logic. Knowing that the best transfer between an inbound and an outbound unit will be when the inbound unit provides exact the amount of products required by the outbound unit. The algorithm will find these units first and link them. The next best step is to find the inbound unit providing most products of that specific type and link them to the outbound units requiring most products of that type. This will lead to the maximum number of products transferred and the biggest change of an optimal link, which is that the remaining amount of products on an inbound unit is equal to the number of products required by the outbound units. The algorithm will therefore be a repetitious process of two steps: finding optimal links and after that finding the best remaining link.

There are certainly more approaches that can be thought of, such as letting all outbound units pass by each inbound unit. This approach would obviously not be the most optimal one to find the best links, because it is quite clear that other approaches seem to be more promising. Therefore, approaches that do not seem to be very promising, like this one, will be let out of consideration. The ones listed above, the MPO, MOL and the PBO focus on one specific point and seem to be promising enough to exceed the performance of baseline approach. Which of these is the most effective one, cannot be determined right away by testing them against each other on a single data set. The batching part, which will be described in the next section can influence the performance of these algorithms. Therefore the linking approaches have to be combined with several batching approaches and tested on multiple data sets to find the most effective total solution.

4.3. Unit batching

The previous section described the linking approaches that seem to be most promising considering the number of links. The literature review revealed that the batching of units at the order picking stations would be another very effective lever to reduce the number of unit visits at the order picking station. This section describes the most promising possible approaches to solving the batching part of the order processing problem. This problem is split into two parts: the first part is finding the most optimal batches of units at both sides of the order picking station and the second part is ordering these

batches in such way that the number of unit visits is reduced as much as possible. The section starts with describing approaches for the first part of the batching problem, which is the second part of the order processing problem. Then these approaches are filtered to find the most promising ones and the chapter finishes with describing the approach for solving the second part of the batching problem.

4.3.1. Batching approaches

The baseline example (appendix B) shows that batching can rather easily be skipped by introducing an order picking configuration of one unit at each side of the station. This makes the algorithms less complex, but at the other hand it misses an opportunity to optimize too. Having several units at each side means that each unit that arrives at the order picking station can transfer its products to multiple other units without adding any extra visits, which reduces the overall number of visits. From a practical point of view, however, many units at the order picking station reduces the speed and the ease of handling. Therefore, the number of units at each side of the order picking station cannot be unlimited, but is limited to three at each side in this research, due to the practical limitations.

Simple Combining Links

The easiest and most straightforward approach to group units into batches, is by taking the list of links and adding the corresponding units to each side of the station until the station is filled at both sides. It is clear that this approach does not perform very much better than the baseline approach, as this will result in two visits per link too. Which means that this approach does not use the possibilities of the extra positions at the order picking station extensively. No algorithms are required in this approach because the order is already defined based on the sequences of the links.

Random Batching

Another straightforward approach is to group the units at one side based on the order in which they are listed in the input unit data. This can be done by every time taking the next n units of that list and grouping them into a batch. Units for the other side of the picking station that are linked to the already assigned units are then added to the batch. If there are more connected units than there is space at the order picking station, then more batches are created having the same units at one side, but other units at the other side. This is done until the full input list of units is assigned to the batches and all corresponding units are assigned too. This method does not find the batches having most links internally, but it does provide a good base for the second part of the batching problem, which is the sequencing of the batches. This is because one side of the picking station is in many batches the same. Which means only one side of the order picking station has to be replaced to get a new batch.

Product Similarity Batching

The previous two approaches are based on the order of the list of links or the order of the input list of units. An alternative could be to batch the units based on similarity of the products on these units. Considering that individual inbound units in distribution warehouses tend to be more homogeneous than individual outbound units, it can be useful to combine the outbound units based on the amount of identical products. This is the aim of the product similarity batching approach (PSB). This can lead to batch where each of the inbound units can provide products to each of the outbound units, which would be a batch that internally is as optimal as possible. To find the units for one side with most products in common an algorithm is required that compares the content of all units. The algorithm for this approach will be a Greedy search algorithm that maximizes the amount of shared products on

multiple units.

To do this, first of all a list is generated with the number of products that can be transferred between each combination of units. Secondly, the combination having most products in common is found. After that, if there are more than two units at the outbound side of the order picking station, the next unit is found that has most products in common with each of the units already found. This is done until all outbound units are grouped. When all groups are found for one side of the batch, the other side of the batches should be filled with units having products in common with any of the units at the first side. The batches can now can be generated in the same way as it is done in the random batching approach, so by creating new batches with the same units at one side and other units at the other side when there are more linked units than positions at the order picking station.

A variation on this approach would be to group the outbound units already before applying the linking process. The group of units, having many products in common, is regarded as one unit in the linking process. Due to the fact that this approach should be applied before solving the first part of the order processing problem, it can be seen as a preprocessing step that can realize better results for the combination of the linking and the batching process. It, however, does not apply a full batching solution, because it only groups one side of the picking station. Therefore another batching approach needs to be applied when implementing this approach before solving the unit linking problem.

Linked Unit Similarity

The PSB approach focuses on batching unit based on the amount of common products. Another approach could be to focus on the amount of linked inbound units that the outbound units have in common. This approach assures that the outbound units have as many as possible combinations with the units at the inbound side of the order picking station. Implementing this Linked Unit Similarity (LUS) approach requires several steps to be taken. First of all a list have to be generated with which outbound units have links with which inbound unit. Then a list have to be generated with how many inbound units each combination of outbound has in common. On that list the Greedy search algorithm will be applied grouping the units that have most products in common. After grouping the outbound units, these groups have to be assigned to batches, together with each of the inbound units that has links with these outbound units. When all links can be handled in one of the batches, the next step is the sequencing of the batches, which is the second part of the batching problem.

Dynamic Changing Batches

All previous described batching approaches generate batches by grouping units, based on for instance the content or linked units. Another approach could be to start with a certain batch, transfer several products, and then replace the units that do not have links with any of the units present at the order picking station, with units that do have combinations with the present units. This way the changing of the units is dynamically, which results every time in the best possible added unit at the order picking station. The downside of this approach is that it can occur that all units that are present at the order picking station do only have a few links left with the units in storage, which will result in inefficient visits where per visit only one product can be transferred.

Implementing this approach falls apart in several parts. The first one is finding a good starting batch, which will be done by iteratively removing the unit with the least links with the others and adding new units having most links to the units present in the batch already. This last part is done by creating a

list showing the amount of links each unit has with the other units and adding up all amounts of the units that are present at the other picking station. The unit having the highest combined amount is the next unit that is chosen. This will be done until a good starting batch is aggregated. The next part is choosing which links should be handled at the order picking station. Which means a certain logic has to be applied to make sure that all links are handled once in a while. This prevents units to be at the other station for a long time and to create new links with fresh units. The last part is to remove the units that do not have links anymore with the other units present in the batch and replace them by the best unit that is available in the storage. This best unit is, like by the creation of the starting batch, chosen based on the number of links with the units already present in the batch.

When a situation occurs in which the units at the order picking station do not have links anymore to the units in the storage, then the inbound units are replaced with the first units in the input data list that still have combinations with other units and the outbound side is replaced by units having most links to the units already assigned to the inbound side.

Every time when a unit is changed, a new batch is created. This means that the order in which the batches are generated is very important and cannot be changed. So the second part of the batching problem is already solved with this Dynamic Changing Batches (DCB) approach.

Dynamic Batching

In the DCB approach the units where changed when they were empty, but it has the drawback of having a change that a situation occurs in which the units present that do not have many links with other units. The Dynamic Batching (DB) solves the batching problem in another way and can probably reduce that drawback. Instead of changing one unit at a time, this approach changes one arbitrary side of the order picking station using a Greedy Search algorithm.
This approach consists of several steps. The first one is the creation of a starting batch, which can be done exactly in the same way as in the DCB approach. The second step is to transfer all links that are possible within the batch and then choose which side should be changed. This is done by first creating two groups of units having most links with the inbound and the outbound side respectively. The group with the highest number of links is then assigned to the order picking station and replaces the unit that were present at that side. This is done, until all units are handled and no links are remaining. When none of the units present at the order picking station does have any link left with the units in storage then one side needs to be changed for the first units at the input unit list that still does have links with other units. Then the other side has to be changed for units having most links with this side. Because of the fixed sequence of the batches that are created, this DB approach solves the second part of the batching problem too. And no further sequencing is to be applied.

4.3.2. Filtering

The seven approaches described above are not equally promising regarding their effectiveness of reducing the number of unit visits at the order picking station. Therefore some filtering has to be applied to select the most promising options. Taking options into account that will definitely be less effective than other options is of no use, as this research focuses on finding the most effective option. The first option that is described above, which is the Simple Combining Links approach, showed already to be equally effective as the baseline approach of not applying batching at all. So this approach will be left out of consideration for the rest of this research, as there are enough other approaches that are

expected to perform better than the baseline approach. The Random Batching approach is expected to perform better than the random batching approach, because of the combining of several unit at once at the other picking station that all can have links. Although this approach is not expected to yield the best results, it is taking into account to check whether and by how much the other options exceed this simple approach.

The Product Similarity Batching approach seems to be very promising, but it has two different moments at which it can be applied. The first one is after the linking process and the other one is before the linking process. Regarding the batching of the units, it actually makes no difference which one of them is applied, because the same units will be grouped before forming the batches. For the linking process, however, it does make a difference because units will now be linked based on the group of units. Therefore the second option seems to be the most promising one and the first option will be left out of consideration. This second option, does, as mentioned, not apply a full batching solution and requires therefore another batching approach to be implemented.
The Linked Unit Similarity approach does seem to be promising, because multiple units will be batched that have a maximum amount of shared linked units. For the Dynamic Changing Batches approach and the Dynamic Batching approach, it is not exactly clear which one will lead to the best results and whether the results will be better than the baseline option. Therefore these two approaches will both be taken into account.

This means that after the filtering of the approaches, five approaches remain. These all need to be tested to find out which one in combination with which linking approach will lead to the best results. Two of the five approaches do not require an additional sequencing step, because sequencing is included already in the approach itself. One approach needs to be performed before the linking approaches and does require another batching approach, because it does not apply a full solution to the batching problem.

4.3.3. Batch Sequencing

Sequencing the generated batches can reduce the number of unit visits required to change from one batch to the next. Starting with one batch, every next batch in the sequence should have the most units in common with the present batch. If there are m batches it means that there are $m!$ options. Therefore again a Nearest Neighbor Greedy Search algorithm is applied to find the order of batches having the least amount of required visits at the transitions between the batches. To find the next best batch, a list is generated showing the amount of transitions required to change from each batch to the next one. Then the best option is taken by evaluating the list. This continues until all batches are put in the sequence. Combining this approach with one of the batching approaches that does not implement sequencing, results in a full solution to the batching problem.
Other approaches then the Batch Sequencing can be applied too, using other algorithms, but this heuristic approach seems to be the most effective one regarding computational as described before. Further research needs to prove which heuristic algorithms can outperform this one in efficiency. This approach will serve as the standard follower after the batching approaches that do not solve the sequencing part themselves. So the Random Batching and the Linked Unit Similarity approach will be followed by this approach to create a full batching solution.

4.4. Simulations

The previous two sections described the approaches to solving the linking and the batching problem respectively, all solving only a part of the full order processing problem. This section describes the implementation of these approaches as distinct modules and the combining of these into simulations. The section starts by describing how the approaches are implemented in code as separate modules. Next it is described how all these modules are combined into simulations to test each of the combinations of solutions addressing the full unit-to-unit order processing problem. It finishes off with the verification and validation of the simulations.

4.4.1. Modules

As described, each of the approaches does only solve a part of the total problem; The linking approaches (MPO, MOL and PBO) do only address the first part. The batching approaches (DCB, DB, RB + BS and LUS + BS) all four create a full solution to the second part of the order processing problem. The PSB approach does not address one of the problem steps at all, but is a preprocessing approach that can be combined with the linking approaches. To test all the combinations of approaches against each other, multiple simulations need to be run. Each of these simulations could be implemented in a separate program, which would mean that each of the approaches needs to be implemented several times in different ways in multiple simulations. Implementing approaches multiple times would waste time, therefore a better approach is chosen in this research, which is implementing all approaches as a separate module. This means ten different modules need to be created: The RL as the baseline, the PSB as the preprocessing step, the MPO, MOL and the PBO as the linking steps, the DCB, DB, RB, LUS and BS as the batching approaches. These modules can then be combined in different ways to create the required simulations.
Combining these modules to come up with a full solution to the unit-to-unit order processing problem can be done in many different ways. But there are certain conditions that each of the combinations of approaches should meet in order to create a valid simulation:

- Each of the order processing problem parts should be addressed exactly once.
- The approaches should address the problem parts in the right order.
- A preprocessing step can be implemented but is not necessary.

Taking these restrictions into account it means that simulations can be started in two ways, with or without the preprocessing module. Then three different linking modules can be applied and after that four batching options can be applied, some consisting of one and others of two modules. This means two times three times four is twenty-four different simulations can be created. Adding the baseline module as a simulation, then it results in twenty-five distinct simulations that should be tested against each other to find out which of these is the best one. Testing the baseline approach for different configurations would result in the same results every time. Therefore, the random batching approach will be added in configurations were batching can be applied to also have a baseline for individual configurations.

Below is listed which modules are part of which simulation. Each simulation is given a number by which they can be identified henceforth.

- (1) RL + RB - Baseline
- (2) MPO + RB + BS
- (3) MPO + LUS + BS
- (4) MPO + DCB
- (5) MPO + DB
- (6) MOL + RB + BS
- (7) MOL + LUS + BS
- (8) MOL + DCB
- (9) MOL + DB
- (10) PBO + RB + BS
- (11) PBO + LUS + BS
- (12) PBO + DCB
- (13) PBO + DB
- (14) PSB + MPO + RB + BS
- (15) PSB + MPO + LUS + BS
- (16) PSB + MPO + DCB
- (17) PSB + MPO + DB
- (18) PSB + MOL + RB + BS
- (19) PSB + MOL + LUS + BS
- (20) PSB + MOL + DCB
- (21) PSB + MOL + DB
- (22) PSB + PBO + RB + BS
- (23) PSB + PBO + LUS + BS
- (24) PSB + PBO + DCB
- (25) PSB + PBO + DB

4.4.2. Implementation

Before implementing each of the approaches as modules in code, they are first developed further in a more in-depth manner as algorithms. The description of each of the algorithms implemented in the module can be found in pseudocode in Appendix C.
To implement these algorithms in real code, a code editor and a language is required. For this research Microsoft Visual Studio 2017 is used as the code editor and C# as the programming language. Visual Studio was chosen because it is used by others in the field of mobile fulfillment systems to test their algorithms[24][25].

During implementation beside the main goal of implementing the model in the right way, two goals were targeted, which are readability and computational speed. This means the code was annotated and naming focused on being as clear and descriptive as possible. Furthermore, it means that the order of each function should be as low as possible and unnecessary computations are reduced to the minimum. Which seem to be obvious, but in practice this results in more complex code affecting readability. Two technologies that showed off being extremely useful in this process is are the Language Integrated Query (LINQ) to objects and MoreLINQ, which is an extension of LINQ. These two libraries were added to Visual Studio, resulting in both increased readability, as the commands are very compact, and in increased computational speed at some points, due to smart execution of multiple

functions at once.

4.4.3. Validation and Verification

To verify the working of each of the modules to make sure that each of them does what it is expected to do, testing is required. To do this a framework is created in which test data is generated and in which each of the modules can be switched on and off. This way the framework creates an testing environment to test and verify each of the modules individually. Testing a module addressing the second part (or third) part of the problem is done by switching on one of the modules addressing the linking option too. This way the output of one module is the input for the next module.

For each algorithm step in each module it can be determined by hand what its output should be based on its input. Then using the debugger of Visual Studio it can be checked whether the output of the program is in line with the expected output. This way each of the algorithm steps in the modules, starting with the modules addressing the linking problem, are tested and debugged exhaustively, using different test sets until all their results where in line with the expectations. Thus it can be concluded that all modules are in line with the model plan, which means that they are verified. The simulations need to be verified too. Knowing that each modules is verified, then for the simulations it is only required to verify that the intended modules for each simulation are executed in the sequence as described for that simulation, which is done using the Visual Studio debugger.

The next step is the validation of the simulations. Validation of the simulations can be done by comparing the results of each of the simulations with each other and with the expected output as was defined in the project outline. If the produced output matches the expected output and the output is a valid solution to the problem corresponding with reality, then this means that it is the right model for solving the problem, which means that the simulation is validated. This is done for all simulations to validate them all.

5

Real case validation and results

Having verified and validated each of the simulations as described in the previous chapter, everything is ready to test the simulations on real data. This real data will be a list of orders of one full day at the distribution warehouse of Royal Lemkes. This chapter first off describes the the input data in little more detail and the context of it. Then it describes the testing plan for this case study to test the algorithms both on robustness and efficiency. Next the results of these test will be presented and the chapter finishes with an analysis and a discussion of the results.

5.1. Input data

This case study is done on the data of the distribution warehouse of Royal Lemkes, which is a leading supplier of plants to large-scale players in the European retail market. This warehouse handles up to one million plants per day on peak days of more than fifteen thousand different kinds. Most of these plants arrive at the same day as they leave. This means that a day for Royal Lemkes starts with an almost empty warehouse. At the beginning of the day it is already clear which products should be picked for which customers, but the exact composition of the inbound units is still undefined. The composition of the outbound units is defined by Royal Lemkes themselves and hence can be known at the beginning of the day.

In the first hours of the day several trucks arrive, delivering units with products from producers, which in this case are plants from growers. When they arrive the composition of the inbound units will be exactly clear. This means that for these units the first orders and visit sequences can be calculated and that the order picking process can start. When making these calculations with too few units, this will lead to many unnecessary visits, as the change of having the best links then, will be low. Therefore the order picking process will start at a certain moment when a significant amount of units is received. The units that arrive after that moment are not yet taken into account, therefore when the group of newly arrived inbound units is large enough, a new calculation is made finding the optimal links between the new and present, but not finished, inbound units and the not yet finished outbound units. These calculations have to be done taking the expedition times of the outbound units into consideration, because when the order picking process is still on-going the first outbound units should leave the warehouse already. This process continues until all orders are finished.

In this research the arrival of inbound units and the expedition times of the outbound units are left out of consideration, because they are regarded as model complexities that do not contribute to answering the research question, as mentioned in the 'Project boundaries' section in chapter 2.

The input data set that was received from Royal Lemkes for this research contains over 590.000 plants, divided over more than 26.000 order lines. Unfortunately the data that was received was a list of order lines and not the expected input data specifying which products are at which inbound unit and which products should be placed at which outbound unit. Therefore, a program is written dividing all products over the units, while taking into account the size of the products and the maximum capacity of a unit. This is done by first sorting on the specified producer and then on the specified customer. For every next product it is checked whether that product fits on this unit or that a new unit needs to be started. Royal Lemkes themselves are applying a better algorithm to sort the plants on the units, which maximizes the total use of the unit capacity. For this research of finding out which approach is best, it does not matter how well the units are filled. But for the overall performance of the algorithms the filling rate and the number of units is important, as better filled units will reduce the number of visits.

All data was received in Microsoft Excel and has to be used in Visual Studio, so a program is written to read the data set in from Excel into Visual Studio, using Office Open XML. Another program is written to filter the input data. This is done because of the fact that not all order lines were complete and that data for another smaller hall was mixed with data for the hall of interest. Which could cause unintended errors. In real-world applications feedback has to be given to the system, but because this is raw data, some filtering is applied to reduce unnecessary and unintended side effects.

Using these programs, data was first of all loaded into Visual Studio, next it was filtered and then the order lines were split over multiple units. This resulted in a list of 22933 order lines specifying 509703 plants. When that list is divided over the in- and outbound units it results in a list of 2900 inbound units and a list of 3073 outbound units. Having a list of inbound and outbound units and the detailed information of the products on these units, it means that the data is in the format that is expected by the simulations.

The expected output of the simulations is a sequenced list of time slots, with per time slot specified which units should be at each side of the picking station. But, as mentioned before, the effectiveness of the simulations will be measured by the computational time and the number of visits. Therefore a few lines of code are added that measure the time each simulation took and the number of unit changes that are required to fill the order picking stations in the way that is specified in the output of the simulations.

5.2. Testing plan

Testing needs to be done according to a certain plan in order to structure the results. The first goal of the tests is to get an insight in how well each simulation performs regarding effectiveness. Which as mentioned consists of two parts, namely the capability of reducing the number of visits to the order picking station and the computational speed. The second goal of the tests is to get an insight in the effect of changing the order picking configuration. This means two groups of tests will be executed.

The first group of tests is executed while varying the order picking configuration. As mentioned before the maximum number of units at each side of the order picking station is restricted to three in this research due to practical limitations. This means that nine configuration options are possible and will be tested, each of them is listed below.

Testing configurations (inbound units / outbound units):

- 1/1
- 2/1
- 3/1
- 1/2
- 2/2
- 3/2
- 1/3
- 2/3
- 3/3

For each of the possible configurations all simulations will be tested, with as input the data set of Royal Lemkes, as described above. As a result, both the number of visits and the computational time will be listed for each combination of simulation and configuration. For the 1/1 configuration no batching can be applied, which means the preprocessing step is useless and so simulation 2 to 13 will have the same results as simulation 14 to 25. Therefore for this first configuration only the first thirteen simulations will be tested. For all other configuration all simulations will be tested, because batching can be applied to all of them. This will result in a table of 213 simulation results, each having specified how much time the computation took and how many visits are required in the proposed solution.

The second group of tests focuses mainly on the influence of the data size on the computational time. This is done by using several data sizes as input for the simulations, from very small to larger then the actual data set. To test all different sizes and get representative results the following data sizes are used as input:

Size of the testing data sets (number of order lines and number of products):

- 100 (1,559 plants)
- 300 (6,308 plants)
- 1,000 (19,036 plants)
- 3,000 (68,538 plants)
- 10,000 (216,084 plants)
- 30,000 (659,959 plants)

Because only one set of data is available from Royal Lemkes and it is difficult to create representative order sets randomly. The data set of different sizes are created by cutting the set into pieces for the smaller sets and by copying the existing data to create extra order lines for the larger sets. The specified number of order lines is the number of lines that remains after applying the filtering of the input data. The configuration for this group of tests will be the 1/1 configuration, because this results in the solutions requiring least computational time, since batching is left out of consideration. It means that 13 simulations will be executed for each of the data sets, which will results in 78 simulation results. For each of them is specified what the generated number of visits is and how much time the calculation takes.

5.3. Algorithm development

Before the testing according to the test plan started, several simulations were executed to get a first impression of the resulting computational time and the number of visits. It worked out that the average

simulation time was around one and a half hour. Running that over two hundred times would take a total computational time of over three hundred hours. Furthermore, one and an half hour of waiting before results are generated is a very long time for warehouse managers willing to start their working day. Therefore, a significant amount of time was spent to reduce the computational speed.

Adding timers to the code showed that the main cause of the slow computations were the large matrices that were created, handled and edited. Reducing their size could significantly reduce the computational time. Checking the content of the matrices, it worked out that a large part of most of the matrices was filled with zeros. Knowing that most of these were useless, each of the modules was edited in such way that the zeros could be eliminated from the matrices.

The Diagnostic Tools in Visual Studio showed that the average percentage of CPU usage was around 12.5%, which corresponds to the usage of only one processor of the CPU. Increasing this usage could yield to faster calculations, therefore time was spent to make as many steps of the algorithms as possible multithreaded. Which means that calculations are split over multiple processors and executed simultaneously, due to which CPU usages can eventually approach 100%. This is done in two ways. The first one is creating threads by hand and split the executions of obvious distinct calculations that can be executed at the same time over multiple cores. The second way is using Parallel LINQ (PLINQ) which is part of the LINQ package. This package creates the opportunity to execute the LINQ statements multithreaded.

Multithreaded execution of some algorithm steps worked out to be slower than single threaded execution. This was many the fact for algorithm steps that did not have to deal with large lists or a large number of computations. In these cases the overhead of splitting the computations into multiple threads and the deciding of which processor should execute which thread, costs more time than that is saved by multithreaded execution. For each of the algorithms it was judged whether their speed would increase by introducing multithreaded execution or not. And it was implemented accordingly.

It is, however, not possible to run all algorithm steps on multiple processors. Some of them are restricted to execution in series due to the fact that the results of each of the previous calculations needs to be taken into account in the next calculation. In other algorithm steps the order of the results is very important. Indexing the results enables some more steps to be multithreaded, but not for all algorithms steps this was possible, which means they are restricted to execution in series.

5.4. Results and discussion

This section describes and discusses the results of the simulations that have been executed following the test plan as defined in the previous section. The results of the group of tests with varying configuration can be found in figure 5.2. For each combination of configuration and simulation this table specifies the computational time and the number of visits required to bring all units to the order picking station.

5.4.1. Results

During execution it became clear that the execution time had dropped significantly by the measures described in the previous chapter. The longest execution time measured for the data set of Royal Lemkens was eight minutes and fifty-four seconds. Furthermore the Diagnostic Tools of Visual Studio showed that most of the modules were actually showing a higher CPU usage than the earlier observed 12.5%. An random example of this is shown in figure 5.1.

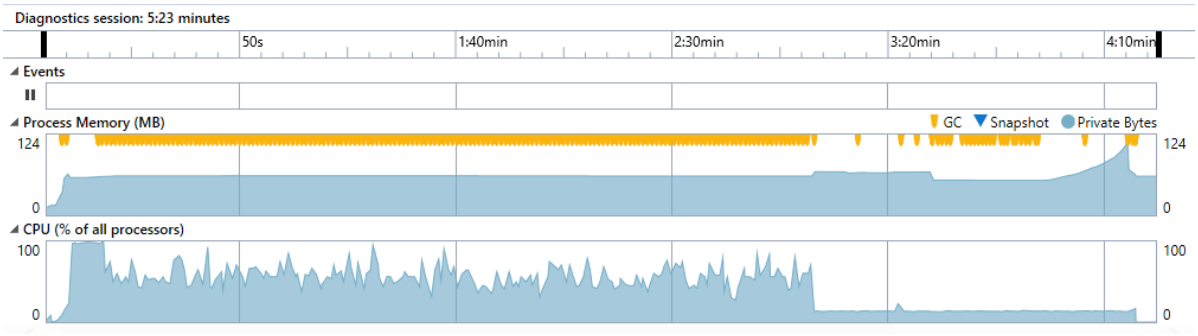

Figure 5.1: CPU and memory usage for simulation 2 in the 3/1 configuration

This figure shows the CPU and memory usage for simulation 2 in the 3/1 configuration, which is a representative one for all other simulations. Most of them show a comparable pattern of fluctuating CPU usage and also a memory usage that does not exceed a few hundred megabytes.

Figure 5.2 shows the results of the first group of tests of the testing plan, in which the configuration was varied. One of the things the table shows is the performance of the baseline algorithm, which is a calculation time of twelve seconds and a total number of 29211 visits required to transfer all products. For the configuration with one inbound unit and one outbound unit several simulations perform better than the baseline, but some of them do not. For this configuration the simulation with the least visits is the second simulation, which requires 23253 visits. This simulation requires four minutes and twenty-three seconds. When taking all different configurations into account the simulation with the least amount of visits is simulation 17 in the 1/3 configuration, with 21874 visits and a computation time of four minutes and twenty-one seconds.
The effectiveness of the algorithms regarding computational time differs from eight seconds to eight minutes and fifty-four second with an average of three minutes and fourteen seconds.

Figure 5.3 shows the results of the second group of tests, in which the size of the input data sets was varied. It shows that the computational time for small data sets is below a second. If the size increases to 10,000 order lines, the computations take for most simulations around half a minute. And when the size is increased to 30,000 order lines, the computational time grows to multiple minutes. It furthermore shows that simulation 3 requires computationally the most time and simulation 12 requires the least computational time averagely.

5.4.2. Analysis and Discussion

The results of both tests show that a certain increase of efficiency regarding the number of unit visits at the order picking station, can be yielded by using another simulation then the baseline one. But

Sim	1/1		1/2		1/3		2/1		2/2		2/3		3/1		3/2		3/3	
1	29221	00:00:12	29063	00:00:12	28858	00:00:14	27943	00:00:13	27858	00:00:14	27761	00:00:14	27281	00:00:13	27218	00:00:13	27156	00:00:13
2	23253	00:04:23	23167	00:03:17	23477	00:03:39	23990	00:04:04	23961	00:03:13	23877	00:03:30	24214	00:04:13	24109	00:03:20	24124	00:03:06
3	23253	00:04:59	23173	00:04:14	23518	00:04:31	22132	00:06:23	22059	00:05:45	22084	00:06:55	21952	00:08:54	21962	00:07:22	21881	00:07:32
4	41969	00:02:47	42935	00:02:51	40048	00:02:58	25875	00:02:37	32221	00:03:01	35976	00:03:15	25791	00:03:24	24780	00:02:50	27892	00:03:21
5	23441	00:03:30	22977	00:03:16	23857	00:03:23	24161	00:02:58	22616	00:03:00	22362	00:03:47	24771	00:03:42	22872	00:03:08	22071	00:03:28
6	23430	00:04:08	23352	00:03:19	23627	00:03:39	24189	00:03:41	24121	00:03:03	24070	00:03:25	24421	00:04:21	24335	00:03:07	24298	00:03:35
7	23430	00:04:44	23340	00:04:03	23663	00:04:36	22357	00:06:22	22270	00:05:44	22239	00:06:24	22170	00:08:45	22166	00:07:35	22062	00:08:08
8	42372	00:02:36	43434	00:02:39	40754	00:03:00	26048	00:02:34	31656	00:02:30	32925	00:02:59	25993	00:03:04	24978	00:02:51	27515	00:02:04
9	23644	00:03:29	23174	00:03:22	24001	00:03:19	24376	00:02:56	22833	00:02:52	22480	00:03:18	24955	00:03:25	23012	00:03:14	22237	00:03:13
10	24311	00:01:46	24224	00:00:55	24493	00:00:50	25017	00:01:17	24942	00:00:39	24831	00:00:34	25190	00:01:37	25078	00:00:42	25037	00:00:27
11	24311	00:02:34	24255	00:01:40	24562	00:01:55	23129	00:03:54	23106	00:03:21	23048	00:03:37	22919	00:06:08	22939	00:05:22	22851	00:05:00
12	45272	00:00:09	46652	00:00:09	44173	00:00:13	27489	00:00:08	32761	00:00:08	33311	00:00:09	27295	00:00:10	26237	00:00:09	27969	00:00:09
13	26405	00:01:04	24068	00:00:38	24943	00:00:36	25350	00:00:36	23645	00:00:33	23356	00:00:31	25966	00:00:38	23906	00:00:30	23056	00:00:25
14			38782	00:04:25	53035	00:05:21	23792	00:04:03	40418	00:03:54	55542	00:05:15	23883	00:05:12	40792	00:04:30	56042	00:04:59
15			38782	00:04:50	53035	00:05:43	23792	00:04:45	36654	00:06:29	49547	00:08:21	23883	00:05:54	36126	00:08:30	48532	00:08:31
16			55930	00:02:55	69754	00:03:26	61009	00:02:39	23587	00:02:39	22612	00:03:47	77450	00:03:52	23614	00:03:07	22623	00:03:50
17			22492	00:03:28	21874	00:04:21	42201	00:03:18	22873	00:03:00	22217	00:04:06	58848	00:04:38	23260	00:03:24	22518	00:03:56
18			39028	00:04:25	53503	00:05:56	23885	00:04:25	40613	00:03:50	55840	00:05:08	23946	00:05:17	40989	00:04:18	56394	00:04:52
19			39028	00:04:25	53503	00:06:10	23885	00:05:26	36849	00:06:11	49786	00:07:10	23946	00:06:03	36277	00:08:29	48785	00:09:25
20			56317	00:02:45	70208	00:03:35	61304	00:02:57	23729	00:02:39	22745	00:03:51	77710	00:03:59	23742	00:03:00	22750	00:03:37
21			22588	00:03:24	21995	00:04:14	42417	00:03:33	22957	00:03:00	22306	00:04:04	59069	00:04:33	23352	00:03:22	22642	00:03:51
22			41084	00:02:05	56739	00:02:48	24671	00:01:43	42539	00:01:40	59036	00:02:18	24658	00:01:48	42804	00:01:46	59242	00:01:59
23			41084	00:02:45	56739	00:03:38	24671	00:02:24	38675	00:04:13	52782	00:04:55	24658	00:02:32	38085	00:05:43	51381	00:05:53
24			61442	00:00:20	77362	00:00:31	64866	00:00:14	25575	00:00:19	24967	00:00:33	82123	00:00:17	25406	00:00:20	24685	00:00:28
25			23667	00:01:00	23111	00:01:13	44055	00:00:58	24020	00:00:39	23456	00:00:55	61631	00:01:01	24449	00:00:36	23739	00:00:40

Figure 5.2: Results of configuration changes (number of visits and computational time - hh:mm:ss)

Sim	100		300		1000		3000		10000		30000	
1	00:00:00	107	00:00:00	340	00:00:00	1134	00:00:00	3346	00:00:02	11389	00:00:19	35956
2	00:00:00	96	00:00:00	313	00:00:00	1081	00:00:03	3185	00:00:37	10565	00:08:00	31647
3	00:00:00	79	00:00:00	277	00:00:00	965	00:00:05	2863	00:01:02	9495	00:15:48	28648
4	00:00:00	96	00:00:00	342	00:00:00	1232	00:00:02	3593	00:00:25	11397	00:05:57	33630
5	00:00:00	90	00:00:00	320	00:00:00	1108	00:00:02	3307	00:00:29	10855	00:06:41	32255
6	00:00:00	96	00:00:00	324	00:00:00	1086	00:00:03	3195	00:00:39	10606	00:07:34	31826
7	00:00:00	80	00:00:00	289	00:00:00	976	00:00:05	2868	00:01:07	9567	00:15:26	28825
8	00:00:00	96	00:00:00	344	00:00:00	1237	00:00:02	3596	00:00:28	11507	00:05:22	33778
9	00:00:00	90	00:00:00	321	00:00:00	1099	00:00:03	3315	00:00:31	10885	00:05:57	32484
10	00:00:00	96	00:00:00	311	00:00:00	1071	00:00:01	3177	00:00:16	10736	00:02:24	33034
11	00:00:00	79	00:00:00	277	00:00:00	969	00:00:03	2860	00:00:41	9679	00:10:30	30163
12	00:00:00	107	00:00:00	353	00:00:00	1236	00:00:00	3655	00:00:02	11829	00:00:11	35672
13	00:00:00	88	00:00:00	321	00:00:00	1126	00:00:01	3371	00:00:07	11147	00:00:49	33934

Figure 5.3: Results of changing data sizes (number of visits and computational time - hh:mm:ss)

this will often go at cost of the efficiency in terms of computational time. This section analyses and discusses the results of both test groups.

Changing configuration

Figure 5.2 shows that the maximum computational time, for the data set of 22933 order lines and more than half a million products, is for none of the combinations of simulations and configuration more than nine minutes. For many applications this can be regarded as short enough and thus computational time can be left out of consideration when choosing which algorithm is most effective. Therefore, the number of visits will mostly determine the effectiveness of the algorithms when the data set is this size or smaller.

When the following three assumptions are true, namely that (1) the configuration of the picking stations can be chosen without restrictions, (2) the configuration does not influence the speed of the order picker and (3) the amount of order lines is comparable or less than the size of this data set, then simulation 17 with configuration 1/3 appears to be the most effective combination. The results of multiple other combinations, like for instance simulation 3 with configuration 3/3, are very close to this one. So to find the exact best solution for a specific warehouse, a range of data sets should be tested before the best combination can be chosen with certainty.

When creating a piece of software that can be used for multiple different customers, it is expected that not all simulations will be implemented, but that only the best one or the best few will be chosen. Because this will reduce the amount of code, lower the risk of bugs and improve the maintainability. This means that, where possible, the simulation with the best overall performance on all configurations will be implemented. Figure 5.4 shows the performance of each of the simulations on all different configurations.

From 5.4 can be seen that many simulations do perform well at some configurations but not on all of them. Simulations 14 up to 25 all have very bad peaks on certain configurations, whereas 2 to 13 do not. This means that the Product Similarity Batching approach does not work out very well on some configurations. The second thing that becomes clear when looking to the figure is that there is a certain pattern in the simulations 2 to 13, which is that three groups of four simulations can be seen, with

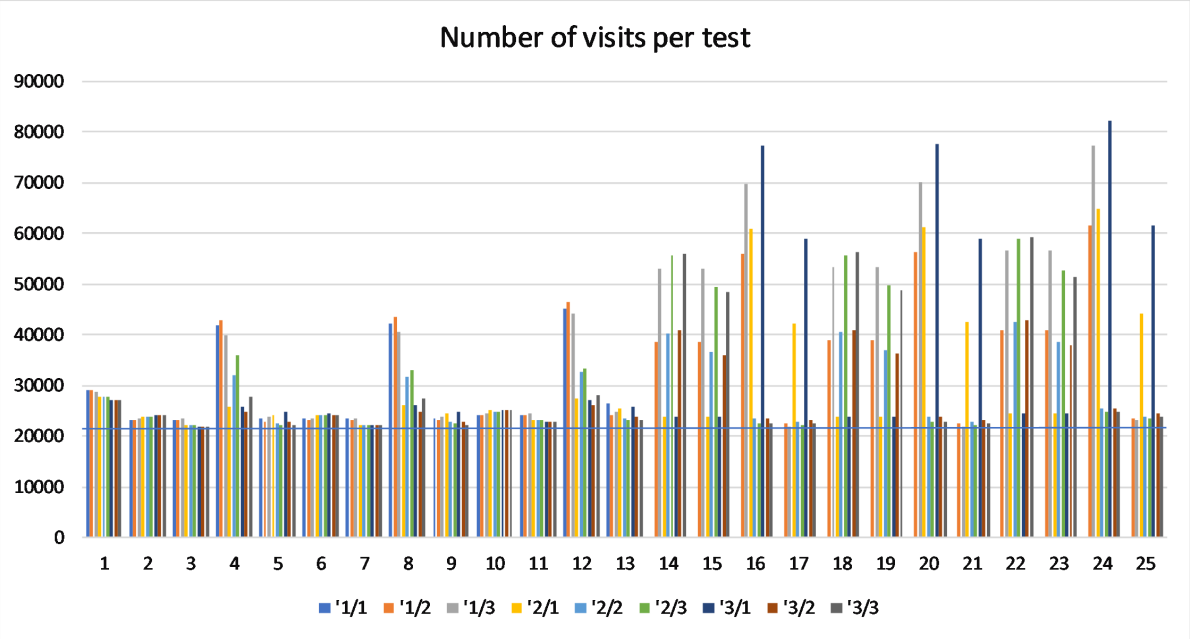

Figure 5.4: Number of visits for each simulation for each order picking station configuration (inbound/outbound units)

very comparable results on all configurations. Within simulations 14 to 25 that same pattern is visible, in which every forth simulation results look like one another. This pattern can easily be interpreted when looking to modules of which each of the simulations is composed; every forth simulation only one module is changed, which does not have a big influence on the results as can be seen in the figure.

Simulation 4, 8 and 12 do not perform very well on the configurations with only a few units at the inbound side, the results are even worse than the baseline. But when more units are applied, results improve a bit. The only common module in these approaches is the Dynamic Changing Batches approach. The bad behaviour on some configurations show that the approach which is at the base of this module is not fittest solution to the problem. The bad behaviour is expected to be caused by the algorithm of finding the best next unit for every empty unit. This algorithm focuses very much on the next best unit for the current situation, which is a too narrow window for order picking stations with small configurations. This results in often changing both sides of the order picking station, where changing only one side would be sufficient. For larger order picking stations this approach works little better, because choosing the best unit then has links with more units, resulting in more picked units per link.

Simulations 14 to 25 all show very bad behaviour on some of the configurations. For simulations 16, 20 and 24 this is expected to be caused by the same reason as simulations 4, 8 and 12, because they all use the DCB module. The results are even worse than for the first three, which can be explained by the fact that due to the batching in the preprocessing step units cannot be changed individually anymore but only in groups, which derails the algorithm, as it is focuses on changing units individually.

The results of 14, 18 and 22 are very much comparable to 15, 19 and 23 respectively. This is due to the fact that the approach of the Random Batching and the Linked Unit Similarity modules is the same when each of the sides of the order picking station is regarded as one and hence no batching can be

applied. In both approaches one side is chosen as static, then the other side is changed until that one side does not have any links left. When the side that is not static is chosen in the preprocessing step and combined into one unit, then this means that the combined side is changed very often. Which logically results in very many unit visits. This can be solved by taking into account which side is already batched in the preprocessing step before choosing a static side. As it can be seen from figure 5.2 and 5.4 that when the batched side is chosen as the static side, then the number of visits can be lower than the baseline result.

Simulations 17, 21 and 25 do perform better than the baseline on many configurations, except on the 2/1 and the 3/1 configuration. The Dynamic Batching module and the preprocessing module do not work together very well on these configurations. Which seems to be obvious when it is noticed that the DB module does not take into account the number of visits that is required to change the batched unit. When this would be taken into account it is expected to lower the number of visits for these configurations too.

From figure 5.2 and 5.4 it can be deducted that for this data set and the described simulations, the simulation with the best average performance on all different configurations is simulation 3. Simulation 7 is very close, but does perform a little less on all configurations.

Changing data set size

Figure 5.3 shows the computational times for each of the simulations having different data set sizes as input. It shows that calculation time is growing to a serious issue for data sizes of 30,000 order lines and 650,000 plants. It can be seen that the simulations that where regarded as the best ones averagely on all configuration are the ones that are computationally requiring the most time. Simulation 3 is taking fifteen minutes and forty-eight seconds and simulation 7 takes fifteen minutes and twenty-six seconds on the data set of 30,000 order lines. Larger data sets will require even more time on these simulations.

With the found times a rough estimation of the order of the simulations can be made. Taking only the last two times into account the order of each of the simulations is shown in the table below. Extrapolating the computational times using the orders shown in the to data sets up to 100,000 lines, results in some computational times exceeding four hours. Which is much longer than the creation of the orders normally takes in most warehouses, which means that it will reduce the throughput time of the orders.

Sim	1	2	3	4	5	6	7	8	9	10	11	12	13
Order	2.83n	3.99n	4.76n	4.43n	4.28n	3.55n	4.27n	3.50n	3.51n	2.67n	4.79n	1.50n	2.00n

The data set size of 100,000, however, will never be reached in the warehouse of Royal Lemkes. Their maximum throughput so far on one day is one million plants. Furthermore, as mentioned before, their warehouse consists of more than one hall, so each hall uses only a subgroup of the total number of plants. This means that the total number of plants in one hall will be less than one million. Another fact that has to be taken into account regarding the maximum order size, will be that not all inbound units are already available when the order picking process starts, but they will arrive during the day.

This means that the process of calculating what unit should visit at which moment at the order picking station should be executed several times a day and only for the units that are present at that moment in the warehouse. The products that have already been transferred can be left out of consideration in these calculations, so the maximum size of the data set that will be calculated at once, is expected to not exceed half a million plants and 25,000 order lines. So the computational time will not grow to a serious issue, which means that it does not influence the simulation choice and the effectiveness of the algorithms.

6

Conclusion and Outlook

6.1. Conclusion

This paper aimed to answer the research question on how the unit-to-unit based order processing problem could be solved as effective as possible for large-scale problems. To be able to do so, first of all the state-of-the-art regarding the order processing problem was reviewed. This resulted in the insight that the state-of-the-art is addressing small scale order processing problems that can be compared with the unit-to-unit order processing problems. Furthermore literature does address large-scale order processing problem in robotic mobile fulfillment systems. Although some of the approaches in literature are pointing in a useful direction, no solutions are presented that can be used for large-scale unit-to-unit order processing problems. Another thing that became clear in the literature study was that no exact solutions could be applied when solving the problems, due to the immense computational time it would lead to, but that a meta-heuristic approach could be a valid alternative.

Now the state-of-the-art was clear and insights are obtained from it, the next step to be able to answer the research question was to investigate which approaches could be generated to solve each of the two parts of the unit-to-unit order processing problem and what should be taken as a baseline. For the linking part of the problem several approaches were generated: Maximizing the Pile-On, Maximizing the Order Line pile-on and Product Based Optimizing. For the batching part of the problem another set of approaches was generated. After filtering the following approaches remained: Random Batching, Product Similarity Batching, Linked Unit Similarity, Dynamic Changing Batches and Dynamic Batching. The Product Similarity Batching was converted to a preprocessing step and the Random Batching and the Linked Unit Similarity both required a sequencing process that was called Batch Sequencing.

The next focus was to transform the approaches into simulations that do address both parts of the order picking problem. To do so, the approaches were first converted to algorithms, which on their turn were converted to modules in code. These modules were combined in all valid ways. This created in total, the baseline included, twenty-five different simulations.

To find out the performance of each of the simulations in terms of effectiveness, they were all tested

on a representative data set, which was provided by Royal Lemkes. Knowing that an order picking station could have multiple different configurations and that the data sets could vary in size per day, all simulations were both tested on each of the configurations and on a number of data set sizes.

The results of both of these tests combined answer the main question for this research, that was stated as follows: How can large-scale unit-to-unit based order processing problems be solved as effective as possible? The tests showed that the computational time does just not play an important role in terms of effectiveness when considering data sets up to 30,000 order lines. Furthermore they showed that, when considering an individual configuration, the combination of the Product Similarity Batching module, the Maximizing Pile-on module and the Dynamic Changing Batching module, works out to be the most effective one, with a configuration of one inbound unit and three outbound units present at the picking station. When choosing one solution to be implemented at different configurations, as Eurobrain is intending to do, the combination of the Maximizing Pile-On module, the Linked Unit Similarity module and the Batch Sequencing module yields the most effective results.

6.2. Outlook

As stated in the discussion of the results, adaptions can be made to some of the modules, which will most probable lead to a further reduction in the number of visits for some simulations on different configurations. Which could, when other solutions would excel the proposed solution in terms of effectiveness, eventually change the recommended solution.

Implementing this solution as a total solution that generates the orders in a warehouse and determines the required sequence unit visits, requires including the complexities that were left out of consideration in this research for the sake of clearness. These complexities are listed below.

- The possibility of having multiple order picking stations was eliminated from the model, but should be included when considering a real warehouse. Most distribution warehouses do have multiple order picking stations. This means that the orders should be split over the stations and the same is true for the units visiting the stations. So instead of one list of orders and one list of unit visits as output, the problem should provide a list of each of the individual order picking stations.
- Starting the order picking process with a subset of the inbound units and redoing the calculations when more inbound units have arrived, enables a longer order picking slot and introduces more flexibility. This is required for at least the case of Royal Lemkes, but multiple other warehouses are expected to have the same wish.
- Taking expedition times into account is another complexity that was left of out consideration, but should be included to enable the shipping process to start while the order picking process is not finished yet. This means some kind of prioritization should be implemented to make sure the units that should be shipped first, can be picked first. This prioritization will interfere with the optimization, therefore a weighted decision has to be made to find the most optimal combinations with the restriction that all outbound units are ready in time.

When each of these complexities is researched and implemented, multiple more complexities can be studied to find out if implementing these would lead to a reduction of the amount of AGVs and the number of order pickers. Examples of these are:

- The routing of one unit and one AGV to multiple order picking stations without returning to storage in between.
- The timing of each of the visits to be performed just in time (JIT), so without waiting to long in buffer positions.
- The optimization of the storage positions to reduce the length of the AGV routes.
- The optimization of the AGV routing to reduce the driving time.
- An optimization of the dispatching of assignments to AGVs.

Summarizing this altogether, it means that a lot of research has to be done before implementation of the unit-to-unit order picking concept is possible. After the implementation much research can be done to increase the efficiency of the system.

Bibliography

[1] N. Boysen, D. Briskorn, and S. Emde, "Parts-to-picker based order processing in a rack-moving mobile robots environment," *European Journal of Operational Research*, vol. 262, no. 2, p. 550–562, 2017.

[2] R. Hamberg and J. Verriet, *Automation in warehouse development*. Springer, 2014.

[3] J. Gu, M. Goetschalckx, and L. Mcginnis, "Research on warehouse operation: A comprehensive review," vol. 177, pp. 1–21, 02 2007.

[4] R. D. Koster, T. Le-Duc, and K. J. Roodbergen, "Design and control of warehouse order picking: A literature review," *European Journal of Operational Research*, vol. 182, no. 2, p. 481–501, 2007.

[5] B. Rouwenhorst, B. Reuter, V. Stockrahm, G. V. Houtum, R. Mantel, and W. Zijm, "Warehouse design and control: Framework and literature review," *European Journal of Operational Research*, vol. 122, no. 3, p. 515–533, 2000.

[6] C. G. Petersen and G. Aase, "A comparison of picking, storage, and routing policies in manual order picking," *International Journal of Production Economics*, vol. 92, no. 1, p. 11–19, 2004.

[7] J. C.-H. Pan and P.-H. Shih, "Evaluation of the throughput of a multiple-picker order picking system with congestion consideration," *Computers Industrial Engineering*, vol. 55, no. 2, p. 379–389, 2008.

[8] J. J. Bartholdi and S. T. Hackman, "Warehouse science," Aug 2017.

[9] J. C.-H. Pan and M.-H. Wu, "Throughput analysis for order picking system with multiple pickers and aisle congestion considerations," *Computers Operations Research*, vol. 39, no. 7, p. 1661–1672, 2012.

[10] V. Khanzode and B. Shah, "A comprehensive review of warehouse operational issues," *International Journal of Logistics Systems and Management*, vol. 26, p. 346, Jan 2017.

[11] M. C. Mountz, "Material handling system and method using mobile autonomous inventory trays and peer-to-peer communications," Sep 2005.

[12] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," in *Proceedings of the 19th National Conference on Innovative Applications of Artificial Intelligence - Volume 2*, IAAI'07, pp. 1752–1759, AAAI Press, 2007.

[13] R. Dandrea and P. Wurman, "Future challenges of coordinating hundreds of autonomous vehicles in distribution facilities," *2008 IEEE International Conference on Technologies for Practical Robot Applications*, 2008.

[14] E. Guizzo, "Three engineers, hundreds of robots, one warehouse," Jul 2008.

[15] J. J. Enright and P. R. Wurman, "Optimization and coordinated autonomy in mobile fulfillment systems," in *Proceedings of the 9th AAAI Conference on Automated Action Planning for Autonomous Mobile Robots*, AAAIWS'11-09, pp. 33–38, AAAI Press, 2011.

[16] "Kiva the disrupter (managing business disruption and ensuring business continuity)," *Strategic Direction*, vol. 29, Dec 2013.

[17] S. Nigam, D. Roy, R. de Koster, and I. Adan, "Analysis of class-based storage strategies for the mobile shelf-based order pick system," in *13TH IMHRC*, 2014.

[18] T. Lamballais, "Inventory allocation in robotic mobile fulfillment systems," *SSRN Electronic Journal*, Jan 2017.

[19] B. Zou, X. Xu, Y. Y. Gong, and R. D. Koster, "Evaluating battery charging and swapping strategies in a robotic mobile fulfillment system," *European Journal of Operational Research*, vol. 267, no. 2, p. 733–753, 2018.

[20] T. Lamballais, D. Roy, and M. D. Koster, "Estimating performance in a robotic mobile fulfillment system," *European Journal of Operational Research*, vol. 256, no. 3, p. 976–990, 2017.

[21] M. Merschformann, L. Xie, and D. Erdmann, "Path planning for robotic mobile fulfillment systems," *CoRR*, vol. abs/1706.09347, 2017.

[22] Z. Yuan and Y. Y. Gong, "Bot-in-time delivery for robotic mobile fulfillment systems," *IEEE Transactions on Engineering Management*, vol. 64, no. 1, p. 83–93, 2017.

[23] X. Xiang, C. Liu, and L. Miao, "Storage assignment and order batching problem in kiva mobile fulfilment system," *Engineering Optimization*, vol. 50, no. 11, p. 1941–1962, 2018.

[24] M. Merschformann, T. Lamballais, R. de Koster, and L. Suhl, "Decision rules for robotic mobile fulfillment systems," *CoRR*, vol. abs/1801.06703, 2018.

[25] M. Merschformann, L. Xie, and H. Li, "Rawsim-o: A simulation framework for robotic mobile fulfillment systems," *CoRR*, vol. abs/1710.04726, 2017.

[26] H. Hwang, W. J. Baek, and M.-K. Lee, "Clustering algorithms for order picking in an automated storage and retrieval system," *International Journal of Production Research*, vol. 26, no. 2, p. 189–201, 2007.

[27] E. A. Elsayed, M.-K. Lee, S. Kim, and E. Scherer, "Sequencing and batching procedures for minimizing earliness and tardiness penalty of order retrievals," *International Journal of Production Research*, vol. 31, no. 3, p. 727–738, 1993.

[28] L. Nicolas, F. Yannick, and H. Ramzi, "Optimization of order batching in a picking system with carousels," *IFAC-PapersOnLine*, vol. 50, no. 1, p. 1106–1113, 2017.

[29] L. Nicolas, F. Yannick, and H. Ramzi, "Order batching in an automated warehouse with several vertical lift modules: Optimization and experiments with real data," *European Journal of Operational Research*, vol. 267, no. 3, p. 958–976, 2018.

[30] G. Improta and G. Cantarella, "Control system design for an individual signalized junction," *Transportation Research Part B: Methodological*, vol. 18, no. 2, p. 147–167, 1984.

[31] H. A. Taha, *Operations research: an introduction*. Pearson Education, 2017.

A

Appendix A: Scientific Research Paper

Abstract

This paper presents multiple solutions to the order processing problem within large scale unit-to-unit order picking systems, which are a new kind of picker-to-parts order picking systems. They can be seen as a generalization and an expansion of the existing robotic mobile fulfillment systems, introduced by Kiva [12]. In these systems, units, like containers and pallets, are brought to stationary order pickers by AGVs, who then transfer the products from one unit to another. The order processing problem concerns the sequencing and batching of the units and the selection of the products to be transferred. After formalizing the problem, solution approaches are proposed, which are implemented in code and tested on a data set of a real warehouse to get insight in their effectiveness. The results show that a significant reduction of AGV fleet size is possible.

A.1. Introduction

In traditional warehousing, using the picker-to-parts systems, the costliest operation is the order picking operation and the costliest activity within the order picking operation is travelling. Reducing the costs for this activity is difficult to accomplish when sticking to the traditional order picking concept. To lower the costs, multiple alternative order picking concepts have been proposed and implemented reducing the amount of labor. But almost any of these systems has the drawback of requiring high investments and lack the scalability of the traditional order picking concept. To resolve these drawbacks, the robotic mobile fulfillment order picking system was introduced by Kiva, improving scalability and requiring less investment. The drawback of this system for most warehouses is that racks have to be developed, replenishment has to take place, orders need to be handled successively and at the end of the line all boxes need to be collected.

Having inbound units with a turnover time of a day or at maximum a few days and an distribution process in which the outbound order sizes are comparable to the inbound order size, another concept can be implemented, which is called the unit-to-unit order picking approach, introduced by Eurobrain. In this concept units like pallets, racks or roller containers, are brought to an order picking station by AGVs. Where products are transferred from one unit to another. This concept seems to be very promising considering the total costs of the order picking operation. This solution, however, can only be implemented when the required investment costs can be reduced to an acceptable level and the return on investment is high. Therefore, research is highly required to reduce the implementation costs of this concept as much as possible in order to contribute to the increasing competitiveness of companies dealing with expensive labor. This research focuses on solving one of them, which is the large scale order processing problem at the order picking station. To do so, it first of all is required to review the state-of-the-art of the order processing problem in mobile fulfillment systems

A.1.1. Literature

The first one in literature presenting the basis of the robotic mobile fulfillment system order picking concept was the founder of Kiva System LLC with patenting parts of the system [11]. In 2007 the three founders of Kiva presented a paper [12] in which they described the Kiva system in certain detail and clarified why this system has more advantages and less drawbacks than the existing systems. In 2008 an article was written providing some more details on how Kiva optimized their assignment of racks. They did it be implementing a multi-agent system, each optimizing their own task by using heuristic methods. [14]

According to Enright & Wurman [15] the biggest lever for reducing the number of robots needed, according to them, is increasing the average number of items picked from each unit that arrives at the order picking station. This concept is then referred to as 'pile-on. They also mention combining orders with many similar products as a lever to increase pile-on. Xiang, Lui & Miao [23] propose batching the orders as a solution to the optimal sequencing of the orders. According to them batching the orders in such way that the similarity within a batch is as high as possible, will gradually reduce the number of unit visits at the order picking station. The order batching problem is NP-hard, so computational time becomes a very important issue when the number of orders increases.

Merschformann et al. [24] propose several different solutions for order sequencing and for unit sequencing. For the order sequencing problem they assume that all picking station are fully filled with orders. When one order is finished a new order has to be chosen from the backlog and assigned to that picking station to take its place. This way no order batching is done, but the decisions are made when an order leaves the station.

Boysen et al. [1] propose three different heuristic approaches to optimize the combination of order sequencing and rack sequencing for one picking station. But, as the computational time increases exponentially on increasing complexity, these algorithms are not useful for the warehouse size addressed in this paper as the computational time should be expressed in years then, instead of seconds.

A.1.2. Discussion

The first insight that is gained from these papers is that solving the order processing problem cannot be done using algorithms finding exact solutions, because they do require an unreasonable amount of computational time for very large problems. As an alternative, meta-heuristic algorithms are used in many papers. The second insight is that there exist two main levers to reduce the number of unit visits at the order picking station: Maximizing the number of products picked for each order from each visiting unit, called pile-on, and batching orders at the order picking station, so that products for multiple orders can be picked at the same time.

A.1.3. Research question

Literature showed that the order processing problem can be split in two parts; the linking of the inbound units to outbound units and the batching of these units at the order picking station, called 'unit linking' and 'unit batching'. In this research the unit linking is called the first part of the unit-to-unit order processing problem and the unit batching is called the second part. If these parts are optimized, the number of AGV visits at the order picking station is reduced, which means that the number of AGVs required for daily operation can be lower, which leads to reduced investment costs.

This optimization problem is NP-hard, as is proved by Boysen et al. [1], which means the number of steps required to solve it cannot be expressed as a polynomial function of the input, so computational time is a serious issue. Therefore, the goal of this research will be to answer the following question:

How can large-scale unit-to-unit based order processing problems be solved as effective as possible?

In which 'effective' means both a minimal number of unit visits to the order picking station and a minimized computational time. Next the focus will be on what approaches can be applied to solve the linking part of the order processing problem and with what approaches the batching part of the problem can be solved. When the possible approaches are clear, the next point is how these can be combined into simulations each of which address both parts of the order processing problem. When that question is addressed and several simulations are generated the last point of interest is which of these simulations is the most effective one, both for a variable data set size and a variable order picking station layout. Therefore the sub-questions to answer the main questions are formulated as follows.

- What approaches can be applied to solve each of the problem parts individually?
- How can these approaches be transformed into simulations addressing the full problem?
- What is the effectiveness of the simulations in terms of the number of visits and computational time?

A.1.4. Project boundaries

To increase clearness of the research it is important to eliminate model complexities that are not required to answer the research question. The most important one that can be left out is time. Finding the optimal number of visits of units at the order picking station can be done without taking time into account. The required number of visits does determine the number of AGVs required and the time it takes to complete all jobs, but the number of AGVs and the time does not have much influence on the number of visits. The second thing that is left out of consideration for this research is the number of order picking stations.

A.2. Methods

The remainder of this paper is structured as follows. First off, for both the linking part and the batching part, several approaches will be generated that are expected to be able to deal with large data sets. These approaches will be transformed into modular algorithms, each of which will at first be verified and validated using a small data set. Then the effectiveness of each of these algorithms will be tested on a specific case, which is a real data set of one day of a distribution warehouse. Information on the performance under different conditions will be gained from tests with multiple data sets and multiple order picking station configurations. The gained results will be discussed and conclusions will be drawn from them.

A.2.1. Baseline

Due to the fact that there is currently no warehouse around in which the unit-to-unit order picking concept is implemented, defining an existing situation as a baseline is not possible. Therefore, in this research the baseline for solving the unit-to-unit order processing problem will be defined as the assembly of the most straightforward approach to solving each individual problem part. Which means that both the linking and the batching process are done randomly. This baseline will be implemented as a simulation to test the performance of other algorithms against this one.

A.2.2. Unit linking

The generated approaches that will be able to solve the first part of the unit-to-unit order processing problem called unit linking, are listed below. An inbound and an outbound unit are denoted as linked when products should be picked from that inbound unit and transferred to that specific outbound unit.

Maximizing Pile-On

The first approach is based on maximizing pile-on (MPO) at the order picking stations, which means that the number of products pick per visit is maximized. This is done by applying a heuristic that is very lightweight regarding required computational performance; the Greedy search algorithm. This is an algorithmic strategy that focuses on finding the next best option at every stage, targeting to approach the global optimum.

Maximizing Order Line pile-on

To find out whether focusing on the amount of products is the best option, or that it is probably better to focus on transferring whole order lines at a time, the second approach focuses on maximizing the number of order lines (MOL) that can be transferred at once. The Greedy Search algorithm is implemented exactly in the same way as is done for the first approach. Every time the products are picked and the number of possible order lines that can be transferred is recalculated. This process repeats itself until the whole data set of provided products on the inbound units and the required units at the outbound units is empty.

Product Based Optimizing

Taking the first best option globally, as the first two examples do, can result in making bad decisions locally. Therefore, it is necessary to weigh up carefully if a local optimization approach would yield better results than the first two approaches. This is done with the third approach, the product based optimizing approach (PBO). This approach focuses on minimizing the number of links per individual product type. This minimization is done using a custom made algorithm, which in short is repetitious process of two steps: finding optimal links and after that finding the best remaining links. When for all types the corresponding links are

found, the next step in the approach will be to combine all these links and remove the duplicates. All links together form a valid approach to the order processing problem in warehouses.

Which of the three presented approaches to the unit linking problem is the most effective one, cannot be determined right away by testing them against each other, because the batching part, which will be described in the next section can influence the performance of these algorithms. Therefore the linking approaches have to be combined with several batching approaches and tested on multiple data sets to find the most effective total solution.

A.2.3. Unit batching

The approaches that will be able to solve the unit batching part of the problem in a better way than the baseline of the unit-to-unit order processing problem, called unit batching, are listed below. This problem is split into two parts: Finding the most optimal batches of units at both sides of the order picking station and ordering these batches in such way that the number of unit visits is reduced as much as possible. The number of units at each side of the order picking station is limited to three at each side in this research, due to the practical limitations.

Random Batching

Another straightforward approach is to group the units at one side based on the order in which they are listed in the input unit data. This can be done by every time taking the next n units of that list and grouping them into a batch. Units for the other side of the picking station that are linked to the already assigned units are then added to the batch. This is done until the full input list of units is assigned to the batches and all corresponding units are assigned too.

Product Similarity Batching

Considering that individual inbound units in distribution warehouses tend to be more homogeneous than individual outbound units, it can be useful to combine the outbound units based on the amount of identical products. This is the aim of the product similarity batching approach (PSB). This can lead to batch where each of the inbound units can provide products to each of the outbound units, which would be a batch that internally is as optimal as possible. The algorithm for this approach will be a Greedy search algorithm that maximizes the amount of shared products on multiple units. This step will be executed before the linking process to realize better results for unit linking part.

Linked Unit Similarity

Another approach is to focus on the amount of linked inbound units that the outbound units have in common. This approach assures that the outbound units have as many as possible combinations with the units at the inbound side of the order picking station. The implementation of this Linked Unit Similarity (LUS) does also make use of the Greedy algorithm in combination with some other algorithm steps.

Dynamic Changing Batches

All previous described batching approaches generate batches by grouping units, based on for instance the content or linked units. Another approach could be to start with a certain batch, transfer several products, and then replace the units that do not have links with any of the units present at the order picking station, with units that do have combinations with the present units. This Dynamic Changing Batches (DCB) approach results every time in the best possible added unit at the order picking station.

Dynamic Batching

In the DCB approach the units where changed when they were empty, but it has the drawback of having a change that a situation occurs in which the units present that do not have many links with other units. The Dynamic Batching (DB) solves the batching problem in another way and can probably reduce that drawback. Instead of changing one unit at a time, this approach changes one arbitrary side of the order picking station using a Greedy Search algorithm.

Batch Sequencing

The PSB and the LUS approach both require an additional step to sequence the batches, because that can reduce the number of unit visits required to change from one batch to the next. Starting with one batch, every next batch in the sequence should have the most units in common with the present batch. If there are m batches it means that there are $m!$ options. Therefore again a Nearest Neighbor Greedy Search algorithm is applied to find the order of batches having the least amount of required visits at the transitions between the batches.

A.2.4. Modules and testing

All mentioned approaches to solving the unit linking and the unit batching part are implemented in code as modules, whilst optimizing computational speed. Combining these modules to come up with a full valid solution to the unit-to-unit order processing problem can be done in 25 different ways as listed below.

- (1) RL + RB - Baseline
- (2) MPO + RB + BS
- (3) MPO + LUS + BS
- (4) MPO + DCB
- (5) MPO + DB
- (6) MOL + RB + BS
- (7) MOL + LUS + BS
- (8) MOL + DCB
- (9) MOL + DB
- (10) PBO + RB + BS
- (11) PBO + LUS + BS
- (12) PBO + DCB
- (13) PBO + DB
- (14) PSB + MPO + RB + BS
- (15) PSB + MPO + LUS + BS
- (16) PSB + MPO + DCB
- (17) PSB + MPO + DB
- (18) PSB + MOL + RB + BS
- (19) PSB + MOL + LUS + BS
- (20) PSB + MOL + DCB
- (21) PSB + MOL + DB
- (22) PSB + PBO + RB + BS
- (23) PSB + PBO + LUS + BS
- (24) PSB + PBO + DCB
- (25) PSB + PBO + DB

All these solutions will be tested for all nine possible configurations at the order picking station with a minimum of one and a maximum of three units at each side.

A.2.5. Validation and Verification

To verify the working of each of the modules to make sure that each of them does what it is expected to do, testing is required. For each algorithm step in each module it can be determined by hand what its output should be based on its input. Then by running the modules it was checked whether their output is in line with the expected output. Thus it can be concluded that all modules are in line with the model plan, which means that they are verified. The simulations need to be verified too. Knowing that each modules is verified, then for the simulations it is only required to verify that the intended modules for each simulation are executed in the sequence as described for that simulation.

The next step is the validation of the simulations. Validation of the simulations can be done by comparing the results of each of the simulations with each other and with the expected output as was defined in the project outline. If the produced output matches the expected output and the output is a valid solution to the problem corresponding with reality, then this means that it is the right model for solving the problem, which means that the simulation is validated. This is done for all simulations to validate them all.

A.3. Results

The results of the tests that have been executed using a data set of an existing warehouse, can be found in figure A.1. For each combination of configuration and solution this graph specifies the number of visits required to bring all units to the order picking station. The very first bar represents the baseline solution, which requires 29211 visits to transfer all products. For this 1/1 configuration the simulation with the least visits is the second simulation, which requires 23253 visits. When taking all different configurations into account the simulation with the least amount of visits is simulation 17 in the 1/3 configuration, with 21874 visits.

Another thing that can be seen from figure A.1 is that not all combinations of solutions and configurations perform better than the baseline solution. This is meanly the case in solution 14 to 25 in which the preprocessing step is applied. This is caused by the fact that a batching approach is applied two times, which often derails the effectiveness of the algorithm. It furthermore can be seen that every fourth solution has very bad peaks too, which means that the Dynamic Changing Batches approach has some unforeseen bad behaviour on some configurations. This is caused by the too narrow optimization window of the algorithm. The effectiveness of the algorithms regarding compu-

tational time differs from eight seconds to eight minutes and fifty-four second with an average of three minutes and fourteen seconds, which means the computational time is low enough to be left out of consideration when choosing the best solution for this example warehouse.

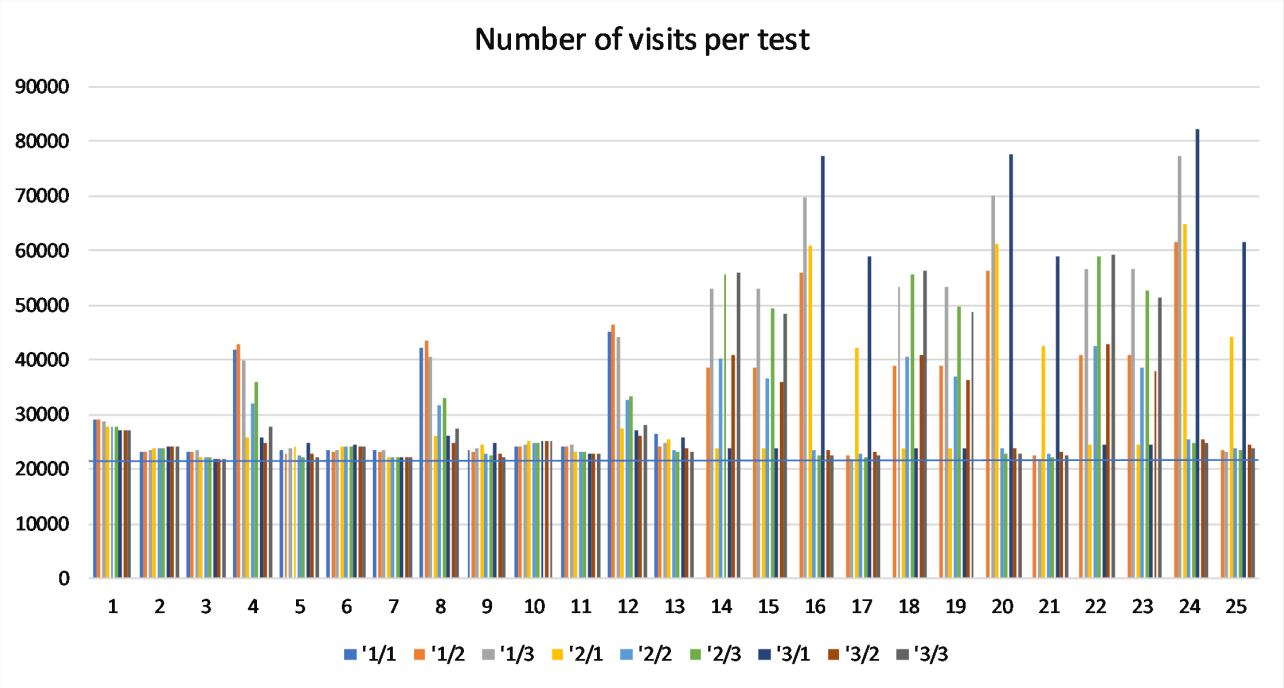

Figure A.1: Number of visits for each simulation for each order picking station configuration (inbound/outbound units)

When the following three assumptions are true, namely that (1) the configuration of the picking stations can be chosen without restrictions, (2) the configuration does not influence the speed of the order picker and (3) the amount of order lines is comparable or less than the size of this data set, then simulation 17 with configuration 1/3 appears to be the most effective combination. When one solution is to be implemented for all different configurations, then solution 3 appears to be the best averagely. But to find the exact best solution for a specific warehouse, a range of data sets should be tested before the best combination can be chosen with certainty. Which answers the research question on how these kind of problem could be solved as effective as possible.

A.4. Discussion

Taking into consideration that it is possible to solve these large-scale order processing problems up to 25% more effective than the baseline approach, it means that the introduction of the introduction of the unit-to-unit order processing problem in reality came a step closer. Other research indicated that optimizing the order processing problem could yield up to 50% better results [1]. This research was based on one sided AGV movement, on a very small data set, with a different baseline, so results can not be compared one-to-one.

Although this research does provide a number of full solutions to the order processing problem in large-scale unit-to-unit order picking systems, much research can be done to improving this result even further. For instance by reducing the bad behaviour some algorithms showed, or by implementing other heuristic approaches. Research should also be done to include the modal complexities that have been excluded in this research, such as time and the number of picking stations.

When considering the unit-to-unit order picking system as a whole, research is also required on the other optimization problems coming with this system. Such as the AGV routing, JIT arrival, assignment dispatching and storage assignment problems. Some of them are already addressed in literature, but many of them got an extra dimension with the introduction of the unit-to-unit order picking system.

B

Appendix B: Example explaining baseline and mathematical model

To clarify the baseline and the mathematical model an example scenario is made up, which is shown in figure B.1. In this example a warehouse has an input of five inbound units I and an output of four outbound units J, providing or requiring five different types of products S. The capacity of both the inbound side C_i and the outbound side C_j of the order picking station is one unit. The first inbound unit called A is considered first, together with the first outbound unit called AA. If products can be transferred from this inbound unit to the outbound unit, then the products that can be picked are saved as orders and removed from the data, the combination of units is saved as a link. Products 1, 2 and 3 can be transferred from A to AA, so A-AA is a link. Then the next outbound unit (BB) is checked, until A is empty. After which the next inbound unit B is considered and again it is checked whether the first outbound unit AA can be filled with products from this unit, then the next outbound unit is checked, until all inbound units are empty and all outbound units are filled. The result of this first part of the baseline is the number of links between the inbound and the outbound units, shown in the table below.

Links	A-AA	B-AA	B-BB	B-CC	C-BB	C-CC	C-DD	D-BB	D-CC	E-BB	E-DD
Products	1,2,3	5	4	5	5	2	5	1	1,2	4	2,4

The last part of the baseline approach defines the sequence in which the combinations of in- and outbound units visit the order picking station. This is done randomly, which for the baseline approach means that the sequence for the order picking station is a copy of the sequence of the list of links. When the first link visits the order picking station, it costs two visits, one for the inbound unit and one for the outbound unit. The second link costs only one visit as the outbound unit is the same as the previous one. The number of unit visits required to fulfill each of the links is shown in the table below.

Links	A-AA	B-AA	B-BB	B-CC	C-BB	C-CC	C-DD	D-BB	D-CC	E-BB	E-DD
Visits	2	1	1	1	2	1	1	2	1	2	1

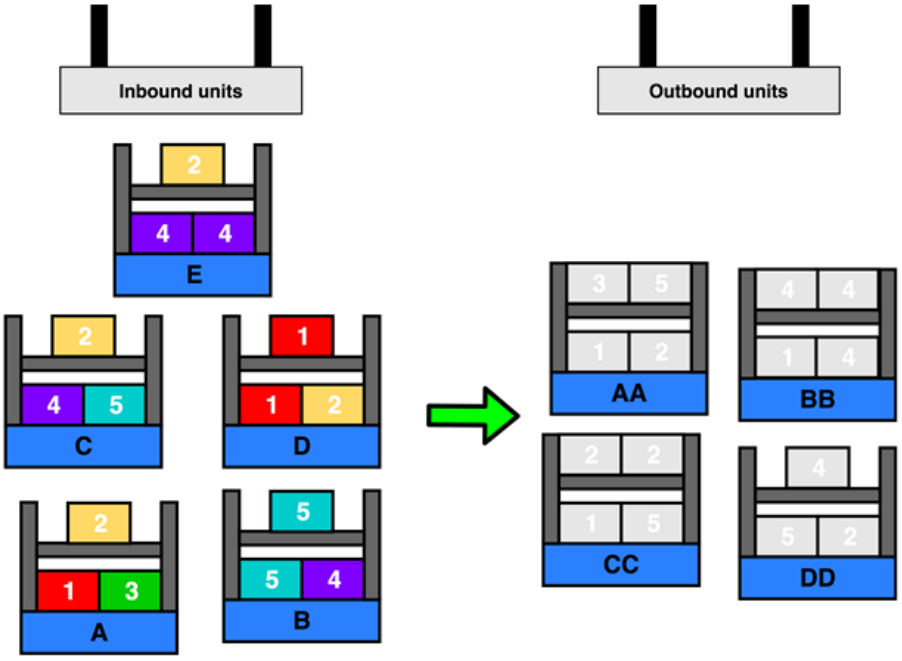

Figure B.1: Example problem

The effectiveness of this algorithm is measured by the time it takes to find all links between the inbound and outbound units and the number of visits it takes to bring all linked units together at the order picking station. If the computational time is low enough, which is based on the specific ideas on it of the warehouse manager, then the number of visits are the main measure of effectiveness. For this example the baseline, the baseline approach leads to an effectiveness of fifteen visits.

C

Appendix C: In-depth algorithms

Algorithm 1: Product Similarity Batching

Data: input data list of outbound units
Result: input data list with grouped outbound units

if *number of outbound units at picking station* $\leq$ *1* **then**
  foreach *combination of unit* **do**
    find number of transferable products;
    store that in a list as percentage of the total content of these outbound units;
  end
  while *list with percentages is not empty* **do**
    create groups of outbound units with highest percentage of products in common;
    groupsize $\leq$ number of units outbound side;
  end
  if *not all outbound units assigned* **then**
    add these to new groups;
  end
  replace input data list of outbound units with the list of groups;
end

Algorithm 2: Random Linking

Data: input data list of in and outbound units
Result: Sequenced list of time slots

foreach *inbound unit* **do**
  foreach *outbound unit* **do**
    if *products can be transferred* **then**
      save the inbound and the outbound unit as a link and delete products temporarily from both units;
    end
  end
end
list of links slots is copied to list of time slots

Algorithm 3: Max Pile-On

Data: input data list of in and outbound units
Result: list of links between inbound and outbound units

foreach *inbound unit* **do**
  foreach *outbound unit* **do**
    number of products that can be transferred is stored in a matrix;
  end
end
find index of highest value in the matrix;
while *value > 0* **do**
  remove the products that can be transferred temporarily from both units and save the link;
  update column and row of index in matrix;
  find index of highest value in the matrix;
end

Algorithm 4: Max Order Line

Data: input data list of in and outbound units
Result: list of links between inbound and outbound units

foreach *inbound unit* **do**
  foreach *outbound unit* **do**
    number of order lines that can be transferred is stored in a matrix;
  end
end
find index of highest value in the matrix;
while *value > 0* **do**
  remove the products that can be transferred temporarily from both units and save the link;
  update column and row of index in matrix;
  find index of highest value in the matrix;
end

Algorithm 5: Product Based Optimizing

Data: input data list of in and outbound units
Result: list of links between inbound and outbound units

```
foreach product type in input data do
    Generate subset of in- and outbound units containing this type and remove all other types
     temporarily;
    while subset is not empty do
        foreach inbound unit in subset do
            foreach outbound unit in subset do
                if number of products on inbound unit = number of products required by
                 outbound unit then
                    save the link and temporarily remove products;
                end
            end
        end
        find inbound unit having most products and outbound unit requiring most products;
        save these as a link and temporarily remove products the number of products that can be
         transferred;
    end
end
```

Algorithm 6: Random Batching

Data: list of links between inbound and outbound units
Result: list of time slots

```
split the inbound units in groups matching the size of the order picking station;
foreach group do
    add group to inbound side of new timeslot;
    foreach unit in group do
        foreach linked outbound unit do
            if outbound side timeslot is not full then
                add link for this unit to outbound side of timeslot;
            else
                add new timeslot with group at inbound side and link at outbound side;
            end
        end
    end
end
```

Algorithm 7: Linked Unit Similarity

Data: list of links between inbound and outbound units
Result: list of time slots

```
if inbound slots order picking station > 1 then
    foreach inbound unit do
        foreach inbound unit do
            Save number of shared outbound units between units in a matrix;
        end
    end
    get combination of units with highest value from matrix;
    while highest value > 0 do
        add this combination to a new group;
        if inbound slot order picking station > 2 then
            while inbound side time slot is not full do
                add inbound unit with highest number of shared outbound units;
            end
        end
        update matrix and get highest value;
    end
end
foreach group do
    add group to inbound side of new timeslot;
    foreach unit in group do
        foreach linked outbound unit do
            if outbound side timeslot is not full then
                add link for this unit to outbound side of timeslot;
            else
                add new timeslot with group at inbound side and link at outbound side;
            end
        end
    end
end
```

Algorithm 8: Dynamic Changing Batches

Data: list of links between inbound and outbound units
Result: sequenced list of time slots

```
if inbound slots order picking station > 1 then
    foreach inbound unit do
        foreach inbound unit do
            save number of shared outbound units between units in a matrix;
        end
    end
    create an initial time slot;
    find highest value in matrix;
    while highest value > 0 do
        transfer all products from next inbound unit;
        replace all units having no links left by units having most links with present units;
        save combination of units as new time slot;
        update matrix and find it's highest value;
    end
end
```

Algorithm 9: Dynamic Batching

Data: list of links between inbound and outbound units
Result: sequenced list of time slots

```
if inbound slots order picking station > 1 then
    foreach link do
        save which combinations each inbound and outbound unit has;
    end
    create an initial time slot;
    while inbound units have links do
        transfer all links at this time slot and update link of in- and outbound lists;
        check replacing which side would result in most new links;
        replace this side and save as new time slot;
    end
end
```

Algorithm 10: Batch Sequencing

Data: list of time slots

Result: sequenced list of time slots

```
if inbound slots order picking station > 1 then
    foreach time slot do
        foreach time slot do
            save the number of unit changes between the time slots in a matrix;
        end
    end
    choose an initial time slot;
    while not all time slots are assigned do
        find slots with same inbound units;
        from these choose the one having most links to other timeslots;
        add the found slots to the sequenced list with the chosen slot as last;
    end
end
```

D

Appendix D: Visual Studio program

```
using Microsoft.Office.Interop.Excel;
using OfficeOpenXml;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;
using MoreLinq.Extensions;

namespace WindowsFormsApp2
{
    public partial class Form1 : Form
    {
        private _Application _excel =
            new Microsoft.Office.Interop.Excel.Application();

        private List<OrderLine> _orderList;
        private List<List<OrderLine>> _inboundUnitList;
        private List<List<OrderLine>> _outboundUnitList;
        private List<int[]> _combinations = new List<int[]>();
        private int _numberOfSlotsInboundSide;
        private int _numberOfSlotsInboundSideInit;
        private int _numberOfSlotsOutboundSide;
        private int _numberOfSlotsOutboundSideInit;
        private List<List<List<int>>> _timeSlots = new List<List<List<int>>>();
        private List<List<List<int>>> _orderedTimeSlots;
        private List<List<int[]>> _linksPerTimeSlot = new List<List<int[]>>();
        private int _visits;
        int _approach1 = -1;
        int _approach1B = -1;
        int _approach2 = -1;
        int _approach3 = -1;
```

```
36          List<List<int>> _combinedInboundUnits = new List<List<int>>();
37          List<List<int>> _combinedOutboundUnits = new List<List<int>>();
38
39          public Form1()
40          {
41              InitializeComponent();
42          }
43
44          private void button1_Click_1(object sender, EventArgs e)
45          {
46              for (int simnumber = 1; simnumber < 14; simnumber++)
47              {
48                  _combinations = new List<int[]>();
49                  _numberOfSlotsInboundSide = 3;
50                  _numberOfSlotsInboundSideInit = _numberOfSlotsInboundSide;
51                  _numberOfSlotsOutboundSide = 1;
52                  _numberOfSlotsOutboundSideInit = _numberOfSlotsOutboundSide;
53                  _timeSlots = new List<List<List<int>>>();
54                  _linksPerTimeSlot = new List<List<int[]>>();
55                  _approach1 = -1;
56                  _approach1B = -1;
57                  _approach2 = -1;
58                  _approach3 = -1;
59                  _combinedInboundUnits = new List<List<int>>();
60                  _combinedOutboundUnits = new List<List<int>>();
61
62                  //int simNumber = Int32.Parse(textBox1.Text);
63                  if (simnumber > 0 && simnumber < 26)
64                  {
65                      switch (simnumber)
66                      {
67                          case 1:
68                              _approach1 = 1;
69                              _approach2 = 5;
70                              _approach3 = -1;
71                              break;
72                          case 2:
73                              _approach1 = 2;
74                              _approach2 = 5;
75                              _approach3 = 9;
76                              break;
77                          case 3:
78                              _approach1 = 2;
79                              _approach2 = 6;
80                              _approach3 = 9;
81                              break;
82                          case 4:
83                              _approach1 = 2;
84                              _approach2 = 7;
85                              _approach3 = -1;
86                              break;
87                          case 5:
88                              _approach1 = 2;
89                              _approach2 = 8;
90                              _approach3 = -1;
91                              break;
92                          case 6:
```

```
93                          _approach1 = 3;
94                          _approach2 = 5;
95                          _approach3 = 9;
96                          break;
97                      case 7:
98                          _approach1 = 3;
99                          _approach2 = 6;
100                         _approach3 = 9;
101                         break;
102                     case 8:
103                         _approach1 = 3;
104                         _approach2 = 7;
105                         _approach3 = -1;
106                         break;
107                     case 9:
108                         _approach1 = 3;
109                         _approach2 = 8;
110                         _approach3 = -1;
111                         break;
112                     case 10:
113                         _approach1 = 4;
114                         _approach2 = 5;
115                         _approach3 = 9;
116                         break;
117                     case 11:
118                         _approach1 = 4;
119                         _approach2 = 6;
120                         _approach3 = 9;
121                         break;
122                     case 12:
123                         _approach1 = 4;
124                         _approach2 = 7;
125                         _approach3 = -1;
126                         break;
127                     case 13:
128                         _approach1 = 4;
129                         _approach2 = 8;
130                         _approach3 = -1;
131                         break;
132                     case 14:
133                         _approach1 = 0;
134                         _approach1B = 2;
135                         _approach2 = 5;
136                         _approach3 = 9;
137                         break;
138                     case 15:
139                         _approach1 = 0;
140                         _approach1B = 2;
141                         _approach2 = 6;
142                         _approach3 = 9;
143                         break;
144                     case 16:
145                         _approach1 = 0;
146                         _approach1B = 2;
147                         _approach2 = 7;
148                         _approach3 = -1;
149                         break;
```

```
150                         case 17:
151                             _approach1 = 0;
152                             _approach1B = 2;
153                             _approach2 = 8;
154                             _approach3 = -1;
155                             break;
156                         case 18:
157                             _approach1 = 0;
158                             _approach1B = 3;
159                             _approach2 = 5;
160                             _approach3 = 9;
161                             break;
162                         case 19:
163                             _approach1 = 0;
164                             _approach1B = 3;
165                             _approach2 = 6;
166                             _approach3 = 9;
167                             break;
168                         case 20:
169                             _approach1 = 0;
170                             _approach1B = 3;
171                             _approach2 = 7;
172                             _approach3 = -1;
173                             break;
174                         case 21:
175                             _approach1 = 0;
176                             _approach1B = 3;
177                             _approach2 = 8;
178                             _approach3 = -1;
179                             break;
180                         case 22:
181                             _approach1 = 0;
182                             _approach1B = 4;
183                             _approach2 = 5;
184                             _approach3 = 9;
185                             break;
186                         case 23:
187                             _approach1 = 0;
188                             _approach1B = 4;
189                             _approach2 = 6;
190                             _approach3 = 9;
191                             break;
192                         case 24:
193                             _approach1 = 0;
194                             _approach1B = 4;
195                             _approach2 = 7;
196                             _approach3 = -1;
197                             break;
198                         case 25:
199                             _approach1 = 0;
200                             _approach1B = 4;
201                             _approach2 = 8;
202                             _approach3 = -1;
203                             break;
204                     }
205                 }
206
```

```
207                 DateTime dt = DateTime.Now;
208
209                 Initialize();
210                 switch (_approach1)
211                 {
212                     case 0:
213                         if (_numberOfSlotsOutboundSide > 1)
214                         {
215                             Sim0BProductSimilarity();
216                         }
217                         else
218                         {
219                             Sim0ProductSimilarity();
220                         }
221
222                         break;
223
224                     case 1:
225                         Sim1RandomLink();
226                         break;
227                     case 2:
228                         Sim2MaxPileOn();
229                         break;
230                     case 3:
231                         Sim3MaxOrderLine();
232                         break;
233                     case 4:
234                         Sim4ProductBasedOptimizing();
235                         break;
236                 }
237
238                 switch (_approach1B)
239                 {
240                     case 2:
241                         Sim2MaxPileOn();
242                         break;
243                     case 3:
244                         Sim3MaxOrderLine();
245                         break;
246                     case 4:
247                         Sim4ProductBasedOptimizing();
248                         break;
249                 }
250
251                 switch (_approach2)
252                 {
253                     case 5:
254                         Sim5RandomBatching();
255                         break;
256                     case 6:
257                         Sim6UnitSimilarity();
258                         break;
259                     case 7:
260                         Sim7DynamicChanging();
261                         break;
262                     case 8:
263                         Sim8DynamicBatching();
```

```
264                     break;
265                 }
266
267                 switch (_approach3)
268                 {
269                     case 0:
270                         break;
271                     case 9:
272                         Sim9MinimumChanges();
273                         break;
274                 }
275
276
277                 if (_approach1B < 0)
278                 {
279                     _visits = _orderedTimeSlots[0][0].Count +
280                               _orderedTimeSlots[0][1].Count;
281                     for (int i = 1; i < _orderedTimeSlots.Count; i++)
282                     {
283                         for (int ii = 0;
284                             ii < _orderedTimeSlots[i][0].Count;
285                             ii++)
286                         {
287                             if (!_orderedTimeSlots[i - 1][0]
288                                 .Contains(_orderedTimeSlots[i][0][ii]))
289                             {
290                                 _visits++;
291                             }
292                         }
293
294                         for (int ii = 0;
295                             ii < _orderedTimeSlots[i][1].Count;
296                             ii++)
297                         {
298                             if (!_orderedTimeSlots[i - 1][1]
299                                 .Contains(_orderedTimeSlots[i][1][ii]))
300                             {
301                                 _visits++;
302                             }
303                         }
304                     }
305                 }
306                 else
307                 {
308                     if (_numberOfSlotsOutboundSideInit > 1)
309                     {
310                         List<int> copyOfOrderdedTimeSlotsOutbound =
311                             new List<int>();
312                         for (int i = 0; i < _orderedTimeSlots.Count; i++)
313                         {
314                             copyOfOrderdedTimeSlotsOutbound.Add(
315                                 _orderedTimeSlots[i][1][0]);
316                         }
317
318                         for (int i = 0; i < _orderedTimeSlots.Count; i++)
319                         {
320                             _orderedTimeSlots[i][1] =
```

```
321                         _combinedOutboundUnits[
322                             copyOfOrderdedTimeSlotsOutbound[i]];
323                     }
324                 }
325                 else
326                 {
327                     List<int> copyOfOrderdedTimeSlotsInbound =
328                         new List<int>();
329                     for (int ii = 0; ii < _orderedTimeSlots.Count; ii++)
330                     {
331                         copyOfOrderdedTimeSlotsInbound.Add(
332                             _orderedTimeSlots[ii][0][0]);
333                     }
334
335                     for (int i = 0; i < _orderedTimeSlots.Count; i++)
336                     {
337                         _orderedTimeSlots[i][0] =
338                             _combinedInboundUnits[
339                                 copyOfOrderdedTimeSlotsInbound[i]];
340                     }
341                 }
342
343                 _visits = _orderedTimeSlots[0][0].Count *
344                           _numberOfSlotsInboundSideInit +
345                           _orderedTimeSlots[0][1].Count;
346                 for (int i = 1; i < _orderedTimeSlots.Count; i++)
347                 {
348                     for (int ii = 0;
349                         ii < _orderedTimeSlots[i][0].Count;
350                         ii++)
351                     {
352                         if (!_orderedTimeSlots[i - 1][0]
353                             .Contains(_orderedTimeSlots[i][0][ii]))
354                         {
355                             _visits++;
356                         }
357                     }
358
359                     for (int ii = 0;
360                         ii < _orderedTimeSlots[i][1].Count;
361                         ii++)
362                     {
363                         if (!_orderedTimeSlots[i - 1][1]
364                             .Contains(_orderedTimeSlots[i][1][ii]))
365                         {
366                             _visits++;
367                         }
368                     }
369                 }
370             }
371
372             for (int i = 0; i < _orderedTimeSlots.Count; i++)
373             {
374                 if (_orderedTimeSlots[i][0].Count >
375                     _numberOfSlotsInboundSideInit ||
376                     _orderedTimeSlots[i][1].Count >
377                     _numberOfSlotsOutboundSideInit)
```

```
                {
                    Console.WriteLine("Error proceeding process");
                }
            }

            textBox2.Text = _visits.ToString() + " " + simnumber;
            DateTime dt2 = DateTime.Now;
            TimeSpan ts = (dt2 - dt);
            textBox3.Text = ts.ToString(@"hh\:mm\:ss");
            Console.WriteLine(simnumber + "    " + _visits.ToString() +
                              "    " + ts.ToString(@"hh\:mm\:ss"));
        }
    }

    //Generating list of inbound units
    private void Initialize()
    {
        var ep = new ExcelPackage(
            new FileInfo(
                @"Filelocation"));
        var ws = ep.Workbook.Worksheets["Sheet1"];

        var domains1 = new List<string>();
        var domains3 = new List<string>();
        var domains4 = new List<string>();
        var domains6 = new List<string>();
        var domains7 = new List<string>();
        var domains8 = new List<string>();
        var domains9 = new List<string>();
        var domains10 = new List<string>();
        var domains11 = new List<string>();

        int temp;
        int count = -1;
        var distinctGrowers = new List<int>();

        _orderList = new List<OrderLine>();
        _inboundUnitList = new List<List<OrderLine>>();

        var inboundUnitFillmentlist = new List<double>();

        for (int rw = 0; rw ≤ ws.Dimension.End.Row; rw++)
        {
            if (ws.Cells[rw + 1, 1].Value == null ||
                ws.Cells[rw + 1, 3].Value == null ||
                ws.Cells[rw + 1, 4].Value == null ||
                ws.Cells[rw + 1, 6].Value == null ||
                ws.Cells[rw + 1, 7].Value == null ||
                ws.Cells[rw + 1, 8].Value == null ||
                ws.Cells[rw + 1, 9].Value == null ||
                ws.Cells[rw + 1, 10].Value == null ||
                ws.Cells[rw + 1, 11].Value == null) continue;
            domains1.Add(ws.Cells[rw + 1, 1].Value.ToString());
            domains3.Add(ws.Cells[rw + 1, 3].Value.ToString());
            domains4.Add(ws.Cells[rw + 1, 4].Value.ToString());
            domains6.Add(ws.Cells[rw + 1, 6].Value.ToString());
            domains7.Add(ws.Cells[rw + 1, 7].Value.ToString());
```

```
435                 domains8.Add(ws.Cells[rw + 1, 8].Value.ToString());
436                 domains9.Add(ws.Cells[rw + 1, 9].Value.ToString());
437                 domains10.Add(ws.Cells[rw + 1, 10].Value.ToString());
438                 domains11.Add(ws.Cells[rw + 1, 11].Value.ToString());
439                 count++;
440
441
442                 if (int.TryParse(domains1[count], out temp) &&
443                     int.TryParse(domains3[count], out temp) &&
444                     int.TryParse(domains4[count], out temp) &&
445                     int.TryParse(domains6[count], out temp) &&
446                     int.TryParse(domains9[count], out temp) &&
447                     int.TryParse(domains10[count], out temp) &&
448                     int.TryParse(domains11[count], out temp))
449                 {
450                     if (int.Parse(domains3[count]) > 0 &&
451                         domains8[count] == "LMK__EXP")
452                     {
453                         _orderList.Add(new OrderLine()
454                         {
455                             CustomerNumber = int.Parse(domains1[count]),
456                             Amount = int.Parse(domains3[count]),
457                             ArticleNumberCustomer = int.Parse(domains4[count]),
458                             Grower = int.Parse(domains6[count]),
459                             DateReceived = domains7[count],
460                             CompanyId = domains8[count],
461                             Layers = int.Parse(domains9[count]),
462                             AmountPerLayer = int.Parse(domains10[count]),
463                             AmountPerBox = int.Parse(domains11[count]),
464                             NumberOfProducts =
465                                 int.Parse(domains3[count]) /
466                                 int.Parse(domains11[count]),
467                             NumberOfProducts2 =
468                                 int.Parse(domains3[count]) /
469                                 int.Parse(domains11[count]),
470                             NumberOfProductsUnused = 0,
471                             NumberOfProductsCounter = 0
472                         });
473                     }
474                 }
475             }
476
477             ThreadStart childRef = new ThreadStart(CallToChildThread);
478
479             Thread childThread = new Thread(childRef);
480             childThread.Start();
481
482             distinctGrowers =
483                 _orderList.Select(x => x.Grower).Distinct().ToList();
484
485             foreach (int element in distinctGrowers)
486             {
487                 var unit = new List<OrderLine>();
488                 var growerOrderList = new List<OrderLine>();
489                 growerOrderList = _orderList.Where(x => x.Grower == element)
490                     .ToList();
491
```

```
492                 double unitFillment = 0.0;
493                 foreach (OrderLine element2 in growerOrderList)
494                 {
495                     Boolean shouldContinue = true;
496                     int amountOfProducts = element2.NumberOfProducts;
497                     int amountPerUnit = 0;
498 
499                     double productSize = (double) 1.0 /
500                                          (double) ((double) element2.Layers *
501                                                    (double) element2
502                                                        .AmountPerLayer);
503                     while (shouldContinue)
504                     {
505                         while ((double) unitFillment + (double) productSize ≤
506                                (double) 1.0000001 &&
507                                amountOfProducts > 0)
508                         {
509                             amountPerUnit++;
510                             amountOfProducts--;
511                             unitFillment += productSize;
512                         }
513 
514                         if (amountPerUnit > 0)
515                         {
516                             element2.NumberOfProducts -= amountPerUnit;
517                             OrderLine tempElement = element2.ShallowCopy();
518                             tempElement.NumberOfProductsUnused = amountPerUnit;
519                             tempElement.NumberOfProductsCounter = amountPerUnit;
520                             unit.Add(tempElement);
521                         }
522 
523                         if (amountOfProducts == 0)
524                         {
525                             shouldContinue = false;
526                         }
527 
528                         if ((double) unitFillment + (double) productSize >
529                             (double) 1.0000001)
530                         {
531                             _inboundUnitList.Add(unit);
532                             unit = new List<OrderLine>();
533                             amountPerUnit = 0;
534                             inboundUnitFillmentlist.Add(unitFillment);
535                             unitFillment = 0;
536                         }
537                     }
538                 }
539 
540                 if (unitFillment > 0)
541                 {
542                     _inboundUnitList.Add(unit);
543                 }
544             }
545 
546             childThread.Join();
547         }
548 
```

```
549         public void CallToChildThread()
550         {
551             var outboundUnitFillmentlist = new List<double>();
552             var distinctklanten = new List<int>();
553             try
554             {
555                 _outboundUnitList = new List<List<OrderLine>>();
556 
557                 distinctklanten = _orderList.Select(x => x.CustomerNumber)
558                     .Distinct().ToList();
559                 foreach (int element in distinctklanten)
560                 {
561                     var unit = new List<OrderLine>();
562                     var klantOrderList = new List<OrderLine>();
563                     klantOrderList = _orderList
564                         .Where(x => x.CustomerNumber == element).ToList();
565 
566                     double unitFillment = 0.0;
567                     foreach (OrderLine element2 in klantOrderList)
568                     {
569                         Boolean shouldContinue = true;
570                         int amountOfProducts = element2.NumberOfProducts2;
571                         int amountPerUnit = 0;
572 
573                         double productSize = (double) 1.0 /
574                                              (double)
575                                              ((double) element2.Layers *
576                                               (double) element2.AmountPerLayer);
577                         while (shouldContinue)
578                         {
579                             while ((double) unitFillment +
580                                    (double) productSize ≤ (double) 1.0000001 &&
581                                    amountOfProducts > 0)
582                             {
583                                 amountPerUnit++;
584                                 amountOfProducts--;
585                                 unitFillment += productSize;
586                             }
587 
588                             if (amountPerUnit > 0)
589                             {
590                                 element2.NumberOfProducts2 -= amountPerUnit;
591                                 OrderLine tempElement = element2.ShallowCopy();
592                                 tempElement.NumberOfProductsUnused =
593                                     amountPerUnit;
594                                 tempElement.NumberOfProductsCounter =
595                                     amountPerUnit;
596                                 unit.Add(tempElement);
597                             }
598 
599                             if (amountOfProducts == 0)
600                             {
601                                 shouldContinue = false;
602                             }
603 
604                             if ((double) unitFillment + (double) productSize >
605                                 (double) 1.0000001)
```

```
606                             {
607                                 _outboundUnitList.Add(unit);
608                                 unit = new List<OrderLine>();
609                                 amountPerUnit = 0;
610                                 outboundUnitFillmentlist.Add(unitFillment);
611                                 unitFillment = 0;
612                             }
613                         }
614                     }
615 
616                     if (unitFillment > 0)
617                     {
618                         _outboundUnitList.Add(unit);
619                     }
620                 }
621             }
622             catch (ThreadAbortException e)
623             {
624                 //Console.WriteLine("Thread Abort Exception");
625             }
626         }
627 
628         private class OrderLine
629         {
630             public int CustomerNumber { get; set; }
631             public int Amount { get; set; }
632             public int ArticleNumberCustomer { get; set; }
633             public int Grower { get; set; }
634             public string DateReceived { get; set; }
635             public string CompanyId { get; set; }
636             public int Layers { get; set; }
637             public int AmountPerLayer { get; set; }
638             public int AmountPerBox { get; set; }
639             public int NumberOfProducts { get; set; }
640             public int NumberOfProducts2 { get; set; }
641             public int NumberOfProductsUnused { get; set; }
642             public int NumberOfProductsCounter { get; set; }
643 
644             public OrderLine ShallowCopy()
645             {
646                 return (OrderLine) this.MemberwiseClone();
647             }
648         }
649 
650         private void Sim0ProductSimilarity()
651         {
652             if (_numberOfSlotsInboundSide > 1)
653             {
654                 List<List<int[]>> inboundSimilarityPercentages =
655                     new List<List<int[]>>();
656                 List<List<int[]>> sortedInboundSimilarityPercentages =
657                     new List<List<int[]>>();
658                 List<List<int[]>> doubleSortedInboundSimilarityPercentages =
659                     new List<List<int[]>>();
660                 List<List<OrderLine>> tempInboundList =
661                     new List<List<OrderLine>>();
662                 _combinedInboundUnits = new List<List<int>>();
```

```
663                 List<OrderLine> tempunit = new List<OrderLine>();
664
665                 for (int i = 0; i < _inboundUnitList.Count - 1; i++)
666                 {
667                     List<int[]> rowOfPercentages = new List<int[]>();
668                     for (int ii = i + 1; ii < _inboundUnitList.Count; ii++)
669                     {
670                         int countSimilarProducts = 0;
671                         int totalNrOfProductsOnUnit = 0;
672                         int totalNrOfProductsOnUnit2 = 0;
673
674                         List<int> tempOrderLineList = new List<int>();
675                         foreach (OrderLine orderline in _inboundUnitList[ii])
676                         {
677                             tempOrderLineList.Add(orderline
678                                 .NumberOfProductsUnused);
679                             totalNrOfProductsOnUnit2 +=
680                                 orderline.NumberOfProductsUnused;
681                         }
682
683                         for (int iii = 0;
684                             iii < _inboundUnitList[i].Count;
685                             iii++)
686                         {
687                             int products = _inboundUnitList[i][iii]
688                                 .NumberOfProductsUnused;
689                             totalNrOfProductsOnUnit += products;
690                             for (int iiii = iii;
691                                 iiii < _inboundUnitList[ii].Count;
692                                 iiii++)
693                             {
694                                 if (products > 0 && tempOrderLineList[iiii] > 0)
695                                 {
696                                     if (_inboundUnitList[i][iii]
697                                             .ArticleNumberCustomer ==
698                                         _inboundUnitList[ii][iiii]
699                                             .ArticleNumberCustomer)
700                                     {
701                                         int similarProducts = Math.Min(products,
702                                             tempOrderLineList[iiii]);
703                                         countSimilarProducts += similarProducts;
704                                         products -= similarProducts;
705                                         tempOrderLineList[iiii] -=
706                                             similarProducts;
707                                     }
708                                 }
709                             }
710                         }
711
712                         if (countSimilarProducts > 0)
713                         {
714                             rowOfPercentages.Add(new int[]
715                             {
716                                 countSimilarProducts * 200 /
717                                 (totalNrOfProductsOnUnit +
718                                  totalNrOfProductsOnUnit2),
719                                 i, ii
```

```
720                         });
721                     }
722                 }
723
724                 if (rowOfPercentages.Count > 0)
725                 {
726                     inboundSimilarityPercentages.Add(rowOfPercentages);
727                 }
728             }
729
730             for (int i = 0; i < inboundSimilarityPercentages.Count - 1; i++)
731             {
732                 sortedInboundSimilarityPercentages.Add(
733                     inboundSimilarityPercentages[i]
734                         .OrderByDescending(x => x[0])
735                         .ToList());
736             }
737
738             doubleSortedInboundSimilarityPercentages =
739                 sortedInboundSimilarityPercentages
740                     .OrderByDescending(x => x.Max(y => y[0])).ToList();
741
742             int slotNr = -1;
743             while (doubleSortedInboundSimilarityPercentages.Count > 0)
744             {
745                 doubleSortedInboundSimilarityPercentages =
746                     doubleSortedInboundSimilarityPercentages
747                         .OrderByDescending(x => x.First()[0]).ToList();
748                 _combinedInboundUnits.Add(new List<int>()
749                 {
750                     doubleSortedInboundSimilarityPercentages[0][0][1],
751                     doubleSortedInboundSimilarityPercentages[0][0][2]
752                 });
753                 slotNr++;
754
755                 if (_numberOfSlotsInboundSide > 2)
756                 {
757                     List<List<int[]>> resultingPercentageLists =
758                         new List<List<int[]>>();
759                     for (int i = 0; i < 2; i++)
760                     {
761                         resultingPercentageLists.Add(
762                             doubleSortedInboundSimilarityPercentages
763                                 .SelectMany(x =>
764                                     x.Where(y =>
765                                         y.Contains(
766                                             _combinedInboundUnits[slotNr]
767                                                 [i]) &&
768                                         y[1] == _combinedInboundUnits[
769                                             slotNr][i] ||
770                                         y[2] == _combinedInboundUnits[
771                                             slotNr][i])).ToList());
772                     }
773
774                     for (int i = 1; i < _numberOfSlotsInboundSide - 1; i++)
775                     {
776                         if (i != 1)
```

```
777                             {
778                                 resultingPercentageLists.Add(
779                                     doubleSortedInboundSimilarityPercentages
780                                         .SelectMany(x =>
781                                             x.Where(y =>
782                                                 y.Contains(
783                                                     _combinedInboundUnits[
784                                                         slotNr][i]) &&
785                                                 y[1] == _combinedInboundUnits[
786                                                     slotNr][i] ||
787                                                 y[2] == _combinedInboundUnits[
788                                                     slotNr][i])).ToList());
789                             }
790 
791                             for (int ii = 0; ii < i + 1; ii++)
792                             {
793                                 for (int iii = 0; iii < ii; iii++)
794                                 {
795                                     resultingPercentageLists[ii].Remove(
796                                         resultingPercentageLists[ii]
797                                             .SingleOrDefault(x =>
798                                                 x.Contains(
799                                                     _combinedInboundUnits[
800                                                         slotNr][iii]) &&
801                                                 x[1] == _combinedInboundUnits[
802                                                     slotNr][iii] ||
803                                                 x[2] == _combinedInboundUnits[
804                                                     slotNr][iii]));
805                                 }
806 
807                                 for (int iii = ii + 1; iii < i + 1; iii++)
808                                 {
809                                     resultingPercentageLists[ii].Remove(
810                                         resultingPercentageLists[ii]
811                                             .SingleOrDefault(x =>
812                                                 x.Contains(
813                                                     _combinedInboundUnits[
814                                                         slotNr][iii]) &&
815                                                 x[1] == _combinedInboundUnits[
816                                                     slotNr][iii] ||
817                                                 x[2] == _combinedInboundUnits[
818                                                     slotNr][iii]));
819                                 }
820                             }
821 
822                             List<int> resultingPercentages = new List<int>();
823                             resultingPercentages.AddRange(
824                                 resultingPercentageLists[0].Select(x => x[0]));
825                             for (int ii = 1; ii < i; ii++)
826                             {
827                                 resultingPercentages = resultingPercentages
828                                     .Zip(resultingPercentageLists[ii],
829                                         (d1, d2) => d1 + d2[0]).ToList();
830                             }
831 
832                             if (resultingPercentages.Count == 0)
833                             {
```

```
834                             break;
835                         }
836
837                         int tempResultingList =
838                             resultingPercentages.IndexOf(
839                                 resultingPercentages.Max());
840                         _combinedInboundUnits[slotNr]
841                             .Add(resultingPercentageLists[0][
842                                 tempResultingList][2]);
843                         _combinedInboundUnits[slotNr]
844                             .Add(resultingPercentageLists[0][
845                                 tempResultingList][1]);
846                         _combinedInboundUnits[slotNr] =
847                             _combinedInboundUnits[slotNr].Distinct()
848                                 .ToList();
849                     }
850                 }
851
852                 for (int i = 0;
853                     i < _combinedInboundUnits[slotNr].Count;
854                     i++)
855                 {
856                     doubleSortedInboundSimilarityPercentages.Select(x =>
857                             x.RemoveAll(y =>
858                                 y[1] == _combinedInboundUnits[slotNr][i] ||
859                                 y[2] == _combinedInboundUnits[slotNr][i]))
860                         .ToList();
861                 }
862
863                 for (int i =
864                         doubleSortedInboundSimilarityPercentages.Count - 1;
865                     i ≥ 0;
866                     i--)
867                 {
868                     if (doubleSortedInboundSimilarityPercentages[i].Count ==
869                         0)
870                     {
871                         doubleSortedInboundSimilarityPercentages
872                             .RemoveAt(i);
873                     }
874                 }
875             }
876
877             List<int> unassignedInboundUnits = new List<int>();
878             for (int i = 0; i < _inboundUnitList.Count; i++)
879             {
880                 unassignedInboundUnits.Add(i);
881             }
882
883             unassignedInboundUnits =
884                 unassignedInboundUnits
885                     .Except(_combinedInboundUnits.SelectMany(x => x))
886                     .ToList();
887
888             for (int i = 0; i < unassignedInboundUnits.Count; i++)
889             {
890                 if (_combinedInboundUnits.Count == 0 ||
```

```
891                         _combinedInboundUnits.Last().Count >=
892                         _numberOfSlotsInboundSide)
893                     {
894                         _combinedInboundUnits.Add(new List<int>());
895                     }
896
897                     _combinedInboundUnits.Last().Add(unassignedInboundUnits[i]);
898                 }
899
900                 for (int i = 0; i < _combinedInboundUnits.Count; i++)
901                 {
902                     tempunit = new List<OrderLine>();
903                     for (int ii = 0; ii < _combinedInboundUnits[i].Count; ii++)
904                     {
905                         for (int iii = 0;
906                             iii < _inboundUnitList[_combinedInboundUnits[i][ii]]
907                                 .Count;
908                             iii++)
909                         {
910                             tempunit.Add(
911                                 _inboundUnitList[_combinedInboundUnits[i][ii]][
912                                     iii]);
913                         }
914                     }
915
916                     tempInboundList.Add(tempunit);
917                 }
918
919                 _inboundUnitList = new List<List<OrderLine>>(tempInboundList);
920             }
921
922             else
923             {
924                 for (int i = 0; i < _inboundUnitList.Count; i++)
925                 {
926                     _combinedInboundUnits.Add(new List<int> {i});
927                 }
928             }
929
930             _numberOfSlotsInboundSide = 1;
931         }
932
933         private void Sim0BProductSimilarity()
934         {
935             if (_numberOfSlotsOutboundSide > 1)
936             {
937                 List<List<int[]>> outboundSimilarityPercentages =
938                     new List<List<int[]>>();
939                 List<List<int[]>> sortedOutboundSimilarityPercentages =
940                     new List<List<int[]>>();
941                 List<List<int[]>> doubleSortedOutboundSimilarityPercentages =
942                     new List<List<int[]>>();
943                 List<List<OrderLine>> tempOutboundList =
944                     new List<List<OrderLine>>();
945                 _combinedOutboundUnits = new List<List<int>>();
946                 List<OrderLine> tempunit = new List<OrderLine>();
947
```

```
948                 for (int i = 0; i < _outboundUnitList.Count - 1; i++)
949                 {
950                     List<int[]> rowOfPercentages = new List<int[]>();
951                     for (int ii = i + 1; ii < _outboundUnitList.Count; ii++)
952                     {
953                         int countSimilarProducts = 0;
954                         int totalNrOfProductsOnUnit = 0;
955                         int totalNrOfProductsOnUnit2 = 0;
956 
957                         List<int> tempOrderLineList = new List<int>();
958                         foreach (OrderLine orderline in _outboundUnitList[ii])
959                         {
960                             tempOrderLineList.Add(orderline
961                                 .NumberOfProductsUnused);
962                             totalNrOfProductsOnUnit2 +=
963                                 orderline.NumberOfProductsUnused;
964                         }
965 
966                         for (int iii = 0;
967                             iii < _outboundUnitList[i].Count;
968                             iii++)
969                         {
970                             int products = _outboundUnitList[i][iii]
971                                 .NumberOfProductsUnused;
972                             totalNrOfProductsOnUnit += products;
973                             for (int iiii = iii;
974                                 iiii < _outboundUnitList[ii].Count;
975                                 iiii++)
976                             {
977                                 if (products > 0 && tempOrderLineList[iiii] > 0)
978                                 {
979                                     if (_outboundUnitList[i][iii]
980                                             .ArticleNumberCustomer ==
981                                         _outboundUnitList[ii][iiii]
982                                             .ArticleNumberCustomer)
983                                     {
984                                         int similarProducts = Math.Min(products,
985                                             tempOrderLineList[iiii]);
986                                         countSimilarProducts += similarProducts;
987                                         products -= similarProducts;
988                                         tempOrderLineList[iiii] -=
989                                             similarProducts;
990                                     }
991                                 }
992                             }
993                         }
994 
995                         if (countSimilarProducts > 0)
996                         {
997                             rowOfPercentages.Add(new int[]
998                             {
999                                 countSimilarProducts * 200 /
1000                                 (totalNrOfProductsOnUnit +
1001                                  totalNrOfProductsOnUnit2),
1002                                 i, ii
1003                             });
1004                         }
```

```
1005                    }
1006
1007                    if (rowOfPercentages.Count > 0)
1008                    {
1009                        outboundSimilarityPercentages.Add(rowOfPercentages);
1010                    }
1011                }
1012
1013                for (int i = 0;
1014                    i < outboundSimilarityPercentages.Count - 1;
1015                    i++)
1016                {
1017                    sortedOutboundSimilarityPercentages.Add(
1018                        outboundSimilarityPercentages[i]
1019                            .OrderByDescending(x => x[0]).ToList());
1020                }
1021
1022                doubleSortedOutboundSimilarityPercentages =
1023                    sortedOutboundSimilarityPercentages
1024                        .OrderByDescending(x => x.Max(y => y[0])).ToList();
1025
1026                int slotNr = -1;
1027                while (doubleSortedOutboundSimilarityPercentages.Count > 0)
1028                {
1029                    doubleSortedOutboundSimilarityPercentages =
1030                        doubleSortedOutboundSimilarityPercentages
1031                            .OrderByDescending(x => x.First()[0]).ToList();
1032                    _combinedOutboundUnits.Add(new List<int>()
1033                    {
1034                        doubleSortedOutboundSimilarityPercentages[0][0][1],
1035                        doubleSortedOutboundSimilarityPercentages[0][0][2]
1036                    });
1037                    slotNr++;
1038
1039                    if (_numberOfSlotsOutboundSide > 2)
1040                    {
1041                        List<List<int[]>> resultingPercentageLists =
1042                            new List<List<int[]>>();
1043                        for (int i = 0; i < 2; i++)
1044                        {
1045                            resultingPercentageLists.Add(
1046                                doubleSortedOutboundSimilarityPercentages
1047                                    .SelectMany(x =>
1048                                        x.Where(y =>
1049                                            y.Contains(
1050                                                _combinedOutboundUnits[slotNr][
1051                                                    i]) &&
1052                                            y[1] == _combinedOutboundUnits[
1053                                                slotNr][i] ||
1054                                            y[2] == _combinedOutboundUnits[
1055                                                slotNr][i])).ToList());
1056                        }
1057
1058                        for (int i = 1; i < _numberOfSlotsOutboundSide - 1; i++)
1059                        {
1060                            if (i != 1)
1061                            {
```

```
1062                        resultingPercentageLists.Add(
1063                            doubleSortedOutboundSimilarityPercentages
1064                                .SelectMany(x =>
1065                                    x.Where(y =>
1066                                        y.Contains(
1067                                            _combinedOutboundUnits[
1068                                                slotNr][i]) &&
1069                                        y[1] == _combinedOutboundUnits[
1070                                            slotNr][i] ||
1071                                        y[2] == _combinedOutboundUnits[
1072                                            slotNr][i])).ToList());
1073                    }
1074
1075                    for (int ii = 0; ii < i + 1; ii++)
1076                    {
1077                        for (int iii = 0; iii < ii; iii++)
1078                        {
1079                            resultingPercentageLists[ii].Remove(
1080                                resultingPercentageLists[ii]
1081                                    .SingleOrDefault(x =>
1082                                        x.Contains(
1083                                            _combinedOutboundUnits[
1084                                                slotNr][iii]) &&
1085                                        x[1] == _combinedOutboundUnits[
1086                                            slotNr][iii] ||
1087                                        x[2] == _combinedOutboundUnits[
1088                                            slotNr][iii]));
1089                        }
1090
1091                        for (int iii = ii + 1; iii < i + 1; iii++)
1092                        {
1093                            resultingPercentageLists[ii].Remove(
1094                                resultingPercentageLists[ii]
1095                                    .SingleOrDefault(x =>
1096                                        x.Contains(
1097                                            _combinedOutboundUnits[
1098                                                slotNr][iii]) &&
1099                                        x[1] == _combinedOutboundUnits[
1100                                            slotNr][iii] ||
1101                                        x[2] == _combinedOutboundUnits[
1102                                            slotNr][iii]));
1103                        }
1104                    }
1105
1106                    List<int> resultingPercentages = new List<int>();
1107                    resultingPercentages.AddRange(
1108                        resultingPercentageLists[0].Select(x => x[0]));
1109                    for (int ii = 1; ii < i; ii++)
1110                    {
1111                        resultingPercentages = resultingPercentages
1112                            .Zip(resultingPercentageLists[ii],
1113                                (d1, d2) => d1 + d2[0]).ToList();
1114                    }
1115
1116                    if (resultingPercentages.Count == 0)
1117                    {
1118                        break;
```

```
1119                        }
1120
1121                        int tempResultingList =
1122                            resultingPercentages.IndexOf(
1123                                resultingPercentages.Max());
1124                        _combinedOutboundUnits[slotNr]
1125                            .Add(resultingPercentageLists[0][
1126                                tempResultingList][2]);
1127                        _combinedOutboundUnits[slotNr]
1128                            .Add(resultingPercentageLists[0][
1129                                tempResultingList][1]);
1130                        _combinedOutboundUnits[slotNr] =
1131                            _combinedOutboundUnits[slotNr].Distinct()
1132                                .ToList();
1133                    }
1134                }
1135
1136                for (int i = 0;
1137                    i < _combinedOutboundUnits[slotNr].Count;
1138                    i++)
1139                {
1140                    doubleSortedOutboundSimilarityPercentages.Select(x =>
1141                            x.RemoveAll(y =>
1142                                y[1] == _combinedOutboundUnits[slotNr][i] ||
1143                                y[2] == _combinedOutboundUnits[slotNr][i]))
1144                        .ToList();
1145                }
1146
1147                for (int i =
1148                        doubleSortedOutboundSimilarityPercentages.Count - 1;
1149                    i ≥ 0;
1150                    i--)
1151                {
1152                    if (doubleSortedOutboundSimilarityPercentages[i]
1153                            .Count == 0)
1154                    {
1155                        doubleSortedOutboundSimilarityPercentages.RemoveAt(
1156                            i);
1157                    }
1158                }
1159            }
1160
1161            List<int> unassignedOutboundUnits = new List<int>();
1162            for (int i = 0; i < _outboundUnitList.Count; i++)
1163            {
1164                unassignedOutboundUnits.Add(i);
1165            }
1166
1167            unassignedOutboundUnits =
1168                unassignedOutboundUnits
1169                    .Except(_combinedOutboundUnits.SelectMany(x => x))
1170                    .ToList();
1171
1172            for (int i = 0; i < unassignedOutboundUnits.Count; i++)
1173            {
1174                if (_combinedOutboundUnits.Count == 0 ||
1175                    _combinedOutboundUnits.Last().Count ≥
```

```
1176                        _numberOfSlotsOutboundSide)
1177                    {
1178                        _combinedOutboundUnits.Add(new List<int>());
1179                    }
1180
1181                    _combinedOutboundUnits.Last()
1182                        .Add(unassignedOutboundUnits[i]);
1183                }
1184
1185                for (int i = 0; i < _combinedOutboundUnits.Count; i++)
1186                {
1187                    tempunit = new List<OrderLine>();
1188                    for (int ii = 0; ii < _combinedOutboundUnits[i].Count; ii++)
1189                    {
1190                        for (int iii = 0;
1191                            iii < _outboundUnitList[
1192                                _combinedOutboundUnits[i][ii]].Count;
1193                            iii++)
1194                        {
1195                            tempunit.Add(
1196                                _outboundUnitList[_combinedOutboundUnits[i][ii]]
1197                                    [iii]);
1198                        }
1199                    }
1200
1201                    tempOutboundList.Add(tempunit);
1202                }
1203
1204                _outboundUnitList = new List<List<OrderLine>>(tempOutboundList);
1205            }
1206
1207            else
1208            {
1209                for (int i = 0; i < _outboundUnitList.Count; i++)
1210                {
1211                    _combinedOutboundUnits.Add(new List<int> {i});
1212                }
1213            }
1214
1215            _numberOfSlotsOutboundSide = 1;
1216        }
1217
1218        private void Sim1RandomLink()
1219        {
1220            int i = -1;
1221            foreach (List<OrderLine> element in _inboundUnitList)
1222            {
1223                i++;
1224                int ii = -1;
1225
1226                foreach (List<OrderLine> element2 in _outboundUnitList)
1227                {
1228                    ii++;
1229                    int teller = 0;
1230
1231                    foreach (OrderLine element4 in element)
1232                    {
```

```
1233                        foreach (OrderLine element3 in element2)
1234                        {
1235                            if (element3.ArticleNumberCustomer ==
1236                                element4.ArticleNumberCustomer)
1237                            {
1238                                teller += Math.Min(
1239                                    element3.NumberOfProductsUnused,
1240                                    element4.NumberOfProductsUnused);
1241                                int overstapelen = Math.Min(
1242                                    element3.NumberOfProductsUnused,
1243                                    element4.NumberOfProductsUnused);
1244                                element3.NumberOfProductsUnused -= overstapelen;
1245                                element4.NumberOfProductsUnused -= overstapelen;
1246                            }
1247                        }
1248                    }
1249
1250                    if (teller > 0)
1251                    {
1252                        int[] k = {i, ii};
1253                        _combinations.Add(k);
1254                    }
1255                }
1256            }
1257
1258            _visits = _combinations.Count * 2;
1259            for (int iii = 1; iii < _combinations.Count; iii++)
1260            {
1261                if (_combinations[iii][0] == _combinations[iii - 1][0])
1262                {
1263                    _visits--;
1264                }
1265
1266                if (_combinations[iii][1] == _combinations[iii - 1][1])
1267                {
1268                    _visits--;
1269                }
1270            }
1271        }
1272
1273        private void Sim2MaxPileOn()
1274        {
1275            List<List<int[]>> inOutAssociation = new List<List<int[]>>();
1276            for (int i = 0; i < _inboundUnitList.Count; i++)
1277            {
1278                inOutAssociation.Add(new List<int[]>());
1279            }
1280
1281            Parallel.For(0, _inboundUnitList.Count, index =>
1282            {
1283                var tempAssociation = new List<int[]>();
1284                for (int i = 0; i < _outboundUnitList.Count; i++)
1285                {
1286                    List<int> tempList1 = new List<int>();
1287                    List<int> tempList2 = new List<int>();
1288
1289                    foreach (OrderLine inboundOrderLine in _inboundUnitList[
```

```
1290                        index])
1291                    {
1292                        tempList1.Add(inboundOrderLine.NumberOfProductsUnused);
1293                    }
1294
1295                    foreach (OrderLine outboundOrderLine in _outboundUnitList[i]
1296                    )
1297                    {
1298                        tempList2.Add(outboundOrderLine.NumberOfProductsUnused);
1299                    }
1300
1301                    int teller = 0;
1302
1303                    for (int ii = 0; ii < _inboundUnitList[index].Count; ii++)
1304                    {
1305                        for (int iii = 0;
1306                            iii < _outboundUnitList[i].Count;
1307                            iii++)
1308                        {
1309                            if (_outboundUnitList[i][iii]
1310                                    .ArticleNumberCustomer ==
1311                                _inboundUnitList[index][ii]
1312                                    .ArticleNumberCustomer)
1313                            {
1314                                teller += Math.Min(tempList2[iii],
1315                                    tempList1[ii]);
1316                                int overstapelen = Math.Min(tempList1[ii],
1317                                    tempList2[iii]);
1318                                tempList1[ii] -= overstapelen;
1319                                tempList2[iii] -= overstapelen;
1320                            }
1321                        }
1322                    }
1323
1324                    if (teller > 0)
1325                    {
1326                        tempAssociation.Add(new int[] {teller, i});
1327                    }
1328                }
1329
1330                inOutAssociation[index] = tempAssociation;
1331            });
1332
1333            int temp = 0;
1334
1335            for (int i = 0; i < inOutAssociation.Count; i++)
1336            {
1337                temp += inOutAssociation[i].Count;
1338            }
1339
1340            var maxValueList = new List<int>();
1341            int totalMaxValue = 1;
1342            while (totalMaxValue > 0)
1343            {
1344                totalMaxValue = 0;
1345                int[] maxIndex = new int[] {0, 0};
1346                for (int i = 0; i < inOutAssociation.Count; i++)
```

```
1347                {
1348                    int maxValue = inOutAssociation[i].Select(x => x[0]).Max();
1349                    if (maxValue > totalMaxValue)
1350                    {
1351                        totalMaxValue = maxValue;
1352                        maxIndex = new int[]
1353                        {
1354                            i,
1355                            inOutAssociation[i]
1356                                .FirstOrDefault(x => x[0] == maxValue)[1]
1357                        };
1358                    }
1359                }
1360
1361                //Transfer products
1362                foreach (OrderLine line2 in _inboundUnitList[maxIndex[0]])
1363                {
1364                    foreach (OrderLine line in _outboundUnitList[maxIndex[1]])
1365                    {
1366                        if (line.ArticleNumberCustomer ==
1367                            line2.ArticleNumberCustomer)
1368                        {
1369                            int overstapelen = Math.Min(
1370                                line.NumberOfProductsUnused,
1371                                line2.NumberOfProductsUnused);
1372                            line.NumberOfProductsUnused -= overstapelen;
1373                            line2.NumberOfProductsUnused -= overstapelen;
1374                        }
1375                    }
1376                }
1377
1378                Parallel.For(0, inOutAssociation.Count, i =>
1379                {
1380                    for (int ii = 0; ii < inOutAssociation[i].Count; ii++)
1381                    {
1382                        if (maxIndex[0] == i ||
1383                            maxIndex[0] == inOutAssociation[i][ii][1] ||
1384                            maxIndex[1] == i ||
1385                            maxIndex[1] == inOutAssociation[i][ii][1])
1386                        {
1387                            List<int> tempList1 = new List<int>();
1388                            List<int> tempList2 = new List<int>();
1389
1390                            foreach (OrderLine inboundOrderLine in
1391                                _inboundUnitList[i])
1392                            {
1393                                tempList1.Add(inboundOrderLine
1394                                    .NumberOfProductsUnused);
1395                            }
1396
1397                            foreach (OrderLine outboundOrderLine in
1398                                _outboundUnitList[inOutAssociation[i][ii][1]])
1399                            {
1400                                tempList2.Add(outboundOrderLine
1401                                    .NumberOfProductsUnused);
1402                            }
1403
```

```
1404                            int teller = 0;
1405                            for (int iii = 0;
1406                                iii < _inboundUnitList[i].Count;
1407                                iii++)
1408                            {
1409                                for (int iv = 0;
1410                                    iv < _outboundUnitList[
1411                                        inOutAssociation[i][ii][1]].Count;
1412                                    iv++)
1413                                {
1414                                    if (_outboundUnitList[
1415                                                inOutAssociation[i][ii][1]][iv]
1416                                            .ArticleNumberCustomer ==
1417                                        _inboundUnitList[i][iii]
1418                                            .ArticleNumberCustomer)
1419                                    {
1420                                        teller += Math.Min(tempList2[iv],
1421                                            tempList1[iii]);
1422                                        int overstapelen =
1423                                            Math.Min(tempList1[iii],
1424                                                tempList2[iv]);
1425                                        tempList1[iii] -= overstapelen;
1426                                        tempList2[iv] -= overstapelen;
1427                                    }
1428                                }
1429                            }
1430
1431                            inOutAssociation[i][ii][0] = teller;
1432                        }
1433                    }
1434                });
1435                _combinations.Add(maxIndex);
1436            }
1437        }
1438
1439        private void Sim3MaxOrderLine()
1440        {
1441            List<List<int[]>> orderLineAssociation = new List<List<int[]>>();
1442            for (int i = 0; i < _inboundUnitList.Count; i++)
1443            {
1444                orderLineAssociation.Add(new List<int[]>());
1445            }
1446
1447            for (int k = 0; k < _inboundUnitList.Count; k++)
1448            {
1449                var tempAssociation = new List<int[]>();
1450                for (int i = 0; i < _outboundUnitList.Count; i++)
1451                {
1452                    List<int> tempList1 = new List<int>();
1453                    List<int> tempList2 = new List<int>();
1454                    foreach (OrderLine inboundOrderLine in _inboundUnitList[k])
1455                    {
1456                        tempList1.Add(inboundOrderLine.NumberOfProductsUnused);
1457                    }
1458
1459                    foreach (OrderLine outboundOrderLine in _outboundUnitList[i]
1460                    )
```

```
1461                    {
1462                        tempList2.Add(outboundOrderLine.NumberOfProductsUnused);
1463                    }
1464
1465                    int teller = 0;
1466
1467
1468                    for (int ii = 0; ii < _inboundUnitList[k].Count; ii++)
1469                    {
1470                        for (int iii = 0;
1471                            iii < _outboundUnitList[i].Count;
1472                            iii++)
1473                        {
1474                            if (_outboundUnitList[i][iii]
1475                                    .ArticleNumberCustomer ==
1476                                _inboundUnitList[k][ii].ArticleNumberCustomer)
1477                            {
1478                                if (Math.Min(
1479                                        _outboundUnitList[i][iii]
1480                                            .NumberOfProductsCounter,
1481                                        _inboundUnitList[k][ii]
1482                                            .NumberOfProductsCounter) > 0)
1483                                {
1484                                    teller += 1;
1485                                    int overstapelen = Math.Min(
1486                                        _inboundUnitList[k][ii]
1487                                            .NumberOfProductsCounter,
1488                                        _outboundUnitList[i][iii]
1489                                            .NumberOfProductsCounter);
1490                                    tempList1[ii] -= overstapelen;
1491                                    tempList2[iii] -= overstapelen;
1492                                }
1493                            }
1494                        }
1495                    }
1496
1497
1498                    if (teller > 0)
1499                    {
1500                        tempAssociation.Add(new int[] {teller, i});
1501                    }
1502                }
1503
1504                orderLineAssociation[k] = tempAssociation;
1505            }
1506
1507            var maxValueList = new List<int>();
1508            int totalmaxvalue = 1;
1509            while (totalmaxvalue > 0)
1510            {
1511                totalmaxvalue = 0;
1512                int[] maxIndex = new int[] {0, 0};
1513                for (int i = 0; i < orderLineAssociation.Count; i++)
1514                {
1515                    int maxValue = orderLineAssociation[i].Select(x => x[0])
1516                        .Max();
1517                    if (maxValue > totalmaxvalue)
```

```
1518                    {
1519                        totalmaxvalue = maxValue;
1520                        maxIndex = new int[]
1521                        {
1522                            i,
1523                            orderLineAssociation[i]
1524                                .FirstOrDefault(x => x[0] == maxValue)[1]
1525                        };
1526                    }
1527                }
1528
1529
1530                //Transfer products
1531                foreach (OrderLine inboundOrderLine in _inboundUnitList[
1532                    maxIndex[0]])
1533                {
1534                    foreach (OrderLine outboundOrderLine in _outboundUnitList[
1535                        maxIndex[1]])
1536                    {
1537                        if (outboundOrderLine.ArticleNumberCustomer ==
1538                            inboundOrderLine.ArticleNumberCustomer)
1539                        {
1540                            int overstapelen = Math.Min(
1541                                outboundOrderLine.NumberOfProductsUnused,
1542                                inboundOrderLine.NumberOfProductsUnused);
1543                            outboundOrderLine.NumberOfProductsUnused -=
1544                                overstapelen;
1545                            inboundOrderLine.NumberOfProductsUnused -=
1546                                overstapelen;
1547                        }
1548                    }
1549                }
1550
1551                Parallel.For(0, orderLineAssociation.Count, i =>
1552                {
1553                    for (int ii = 0; ii < orderLineAssociation[i].Count; ii++)
1554                    {
1555                        if (maxIndex[0] == i ||
1556                            maxIndex[0] == orderLineAssociation[i][ii][1] ||
1557                            maxIndex[1] == i ||
1558                            maxIndex[1] == orderLineAssociation[i][ii][1])
1559                        {
1560                            List<int> tempList1 = new List<int>();
1561                            List<int> tempList2 = new List<int>();
1562
1563                            foreach (OrderLine inboundOrderLine in
1564                                _inboundUnitList[i])
1565                            {
1566                                tempList1.Add(inboundOrderLine
1567                                    .NumberOfProductsUnused);
1568                            }
1569
1570                            foreach (OrderLine outboundOrderLine in
1571                                _outboundUnitList[
1572                                    orderLineAssociation[i][ii][1]])
1573                            {
1574                                tempList2.Add(outboundOrderLine
```

```
1575                                    .NumberOfProductsUnused);
1576                            }
1577
1578                            int teller = 0;
1579                            for (int iii = 0;
1580                                iii < _inboundUnitList[i].Count;
1581                                iii++)
1582                            {
1583                                for (int iv = 0;
1584                                    iv < _outboundUnitList[
1585                                        orderLineAssociation[i][ii][1]].Count;
1586                                    iv++)
1587                                {
1588                                    if (_outboundUnitList[
1589                                                orderLineAssociation[i][ii][1]][
1590                                                iv]
1591                                            .ArticleNumberCustomer ==
1592                                        _inboundUnitList[i][iii]
1593                                            .ArticleNumberCustomer)
1594                                    {
1595                                        teller += Math.Min(tempList2[iv],
1596                                            tempList1[iii]);
1597                                        int overstapelen =
1598                                            Math.Min(tempList1[iii],
1599                                                tempList2[iv]);
1600                                        tempList1[iii] -= overstapelen;
1601                                        tempList2[iv] -= overstapelen;
1602                                    }
1603                                }
1604                            }
1605
1606                            orderLineAssociation[i][ii][0] = teller;
1607                        }
1608                    }
1609                });
1610
1611                _combinations.Add(maxIndex);
1612            }
1613        }
1614
1615        private void Sim4ProductBasedOptimizing()
1616        {
1617            List<int> distinctproducts = _orderList
1618                .Select(x => x.ArticleNumberCustomer).Distinct().ToList();
1619            foreach (int element in distinctproducts)
1620            {
1621                List<int[]> subSetInboundList =
1622                    CreateSubSetList(element, _inboundUnitList);
1623                List<int[]> subSetOutboundList =
1624                    CreateSubSetList(element, _outboundUnitList);
1625
1626                for (int i = subSetInboundList.Count - 1; i ≥ 0; i--)
1627                {
1628                    for (int ii = subSetOutboundList.Count - 1; ii ≥ 0; ii--)
1629                    {
1630                        //If the number of products is equal for the inbound
1631                        //and outbound subset list unit, a combination is found.
```

```
1632                    if (subSetInboundList[i][0] ==
1633                        subSetOutboundList[ii][0])
1634                    {
1635                        _combinations.Add(new int[]
1636                        {
1637                            subSetInboundList[i][1],
1638                            subSetOutboundList[ii][1]
1639                        });
1640                        int teller = 0;
1641                        foreach (OrderLine element4 in _inboundUnitList[
1642                            subSetInboundList[i][1]])
1643                        {
1644                            foreach (OrderLine element3 in _outboundUnitList
1645                                [subSetOutboundList[ii][1]])
1646                            {
1647                                if (element3.ArticleNumberCustomer ==
1648                                    element4.ArticleNumberCustomer)
1649                                {
1650                                    teller += Math.Min(
1651                                        element3.NumberOfProductsUnused,
1652                                        element4.NumberOfProductsUnused);
1653                                    int overstapelen = Math.Min(
1654                                        element3.NumberOfProductsUnused,
1655                                        element4.NumberOfProductsUnused);
1656                                    element3.NumberOfProductsUnused -=
1657                                        overstapelen;
1658                                    element4.NumberOfProductsUnused -=
1659                                        overstapelen;
1660                                }
1661                            }
1662                        }
1663
1664                        subSetInboundList.RemoveAt(i);
1665                        subSetOutboundList.RemoveAt(ii);
1666                        break;
1667                    }
1668                }
1669            }
1670
1671            while (subSetInboundList.Count > 0 &&
1672                   subSetOutboundList.Count > 0)
1673            {
1674                subSetInboundList = subSetInboundList
1675                    .OrderByDescending(x => x[0]).ToList();
1676                subSetOutboundList = subSetOutboundList
1677                    .OrderByDescending(x => x[0]).ToList();
1678                if (subSetInboundList.Count > 0)
1679                {
1680                    if (subSetInboundList[0][0] > subSetOutboundList[0][0])
1681                    {
1682                        subSetInboundList[0][0] -= subSetOutboundList[0][0];
1683                        _combinations.Add(new int[]
1684                        {
1685                            subSetInboundList[0][1],
1686                            subSetOutboundList[0][1]
1687                        });
1688                        int teller = 0;
```

```
1689                    foreach (OrderLine element4 in _inboundUnitList[
1690                        subSetInboundList[0][1]])
1691                    {
1692                        foreach (OrderLine element3 in _outboundUnitList
1693                            [subSetOutboundList[0][1]])
1694                        {
1695                            if (element3.ArticleNumberCustomer ==
1696                                element4.ArticleNumberCustomer)
1697                            {
1698                                teller += Math.Min(
1699                                    element3.NumberOfProductsUnused,
1700                                    element4.NumberOfProductsUnused);
1701                                int overstapelen = Math.Min(
1702                                    element3.NumberOfProductsUnused,
1703                                    element4.NumberOfProductsUnused);
1704                                element3.NumberOfProductsUnused -=
1705                                    overstapelen;
1706                                element4.NumberOfProductsUnused -=
1707                                    overstapelen;
1708                            }
1709                        }
1710                    }
1711
1712                    subSetOutboundList.RemoveAt(0);
1713                    if (subSetOutboundList.Count == 0 &&
1714                        subSetInboundList.Count != 0)
1715                    {
1716                        subSetInboundList.RemoveAt(0);
1717                    }
1718                    else
1719                    {
1720                        for (int i = 0;
1721                            i < subSetInboundList.Count;
1722                            i++)
1723                        {
1724                            if (subSetInboundList[i][0] ==
1725                                subSetOutboundList[0][0])
1726                            {
1727                                _combinations.Add(new int[]
1728                                {
1729                                    subSetInboundList[i][1],
1730                                    subSetOutboundList[0][1]
1731                                });
1732                                teller = 0;
1733                                foreach (OrderLine element4 in
1734                                    _inboundUnitList[
1735                                        subSetInboundList[i][1]])
1736                                {
1737                                    foreach (OrderLine element3 in
1738                                        _outboundUnitList[
1739                                            subSetOutboundList[0][1]])
1740                                    {
1741                                        if (element3
1742                                                .ArticleNumberCustomer ==
1743                                            element4
1744                                                .ArticleNumberCustomer)
1745                                        {
```

```
1746                                        teller += Math.Min(
1747                                            element3
1748                                                .NumberOfProductsUnused,
1749                                            element4
1750                                                .NumberOfProductsUnused);
1751                                        int overstapelen = Math.Min(
1752                                            element3
1753                                                .NumberOfProductsUnused,
1754                                            element4
1755                                                .NumberOfProductsUnused);
1756                                        element3
1757                                                .NumberOfProductsUnused
1758                                            -= overstapelen;
1759                                        element4
1760                                                .NumberOfProductsUnused
1761                                            -= overstapelen;
1762                                    }
1763                                }
1764                            }
1765
1766                            subSetInboundList.RemoveAt(i);
1767                            subSetOutboundList.RemoveAt(0);
1768
1769                            break;
1770                        }
1771                    }
1772                }
1773            }
1774            else
1775            {
1776                subSetOutboundList[0][0] -= subSetInboundList[0][0];
1777                _combinations.Add(new int[]
1778                {
1779                    subSetInboundList[0][1],
1780                    subSetOutboundList[0][1]
1781                });
1782                int teller = 0;
1783                foreach (OrderLine element4 in _inboundUnitList[
1784                    subSetInboundList[0][1]])
1785                {
1786                    foreach (OrderLine element3 in _outboundUnitList
1787                        [subSetOutboundList[0][1]])
1788                    {
1789                        if (element3.ArticleNumberCustomer ==
1790                            element4.ArticleNumberCustomer)
1791                        {
1792                            teller += Math.Min(
1793                                element3.NumberOfProductsUnused,
1794                                element4.NumberOfProductsUnused);
1795                            int overstapelen = Math.Min(
1796                                element3.NumberOfProductsUnused,
1797                                element4.NumberOfProductsUnused);
1798                            element3.NumberOfProductsUnused -=
1799                                overstapelen;
1800                            element4.NumberOfProductsUnused -=
1801                                overstapelen;
1802                        }
```

```
1803                        }
1804                    }
1805
1806                    subSetInboundList.RemoveAt(0);
1807                    if (subSetOutboundList.Count != 0 &&
1808                        subSetInboundList.Count == 0)
1809                    {
1810                        subSetOutboundList.RemoveAt(0);
1811                    }
1812                    else
1813                    {
1814                        for (int i = 0;
1815                            i < subSetOutboundList.Count;
1816                            i++)
1817                        {
1818                            if (subSetOutboundList[i][0] ==
1819                                subSetInboundList[0][0])
1820                            {
1821                                _combinations.Add(new int[]
1822                                {
1823                                    subSetInboundList[0][1],
1824                                    subSetOutboundList[i][1]
1825                                });
1826                                teller = 0;
1827                                foreach (OrderLine element4 in
1828                                    _inboundUnitList[
1829                                        subSetInboundList[0][1]])
1830                                {
1831                                    foreach (OrderLine element3 in
1832                                        _outboundUnitList[
1833                                            subSetOutboundList[i][1]])
1834                                    {
1835                                        if (element3
1836                                                .ArticleNumberCustomer ==
1837                                            element4
1838                                                .ArticleNumberCustomer)
1839                                        {
1840                                            teller += Math.Min(
1841                                                element3
1842                                                    .NumberOfProductsUnused,
1843                                                element4
1844                                                    .NumberOfProductsUnused);
1845                                            int overstapelen = Math.Min(
1846                                                element3
1847                                                    .NumberOfProductsUnused,
1848                                                element4
1849                                                    .NumberOfProductsUnused);
1850                                            element3
1851                                                    .NumberOfProductsUnused
1852                                                -= overstapelen;
1853                                            element4
1854                                                    .NumberOfProductsUnused
1855                                                -= overstapelen;
1856                                        }
1857                                    }
1858                                }
1859
```

```
1860                                        subSetOutboundList.RemoveAt(i);
1861                                        subSetInboundList.RemoveAt(0);
1862                                        break;
1863                                    }
1864                                }
1865                            }
1866                        }
1867                    }
1868                }
1869            }
1870        }
1871
1872        private void Sim5RandomBatching()
1873        {
1874            //Make list to find related outbound units for each inbound unit
1875            List<List<int>> tempInboundOutboundList = new List<List<int>>();
1876            for (int i = 0; i < _inboundUnitList.Count; i++)
1877            {
1878                tempInboundOutboundList.Add(_combinations.Where(x => x[0] == i)
1879                    .Select(x => x[1]).ToList());
1880            }
1881
1882            //For every succeeding group of inbound units
1883            List<int> oneTimeSlotInboundUnits = new List<int>();
1884            List<int> oneTimeSlotOutboundUnits = new List<int>();
1885            List<List<int[]>> combinationsPerTimeSlot = new List<List<int[]>>();
1886
1887            for (int i = 0;
1888                i < (_inboundUnitList.Count + _numberOfSlotsInboundSide - 1) /
1889                _numberOfSlotsInboundSide;
1890                i++)
1891            {
1892                //Get the range of inbound unit that will be added to each timeslot
1893                List<int> outboundscombined = new List<int>();
1894                oneTimeSlotInboundUnits = new List<int>();
1895                for (int ii = 0;
1896                    ii < _numberOfSlotsInboundSide &&
1897                    i * _numberOfSlotsInboundSide + ii < _inboundUnitList.Count;
1898                    ii++)
1899                {
1900                    oneTimeSlotInboundUnits.Add(
1901                        i * _numberOfSlotsInboundSide + ii);
1902                    outboundscombined.AddRange(
1903                        tempInboundOutboundList[
1904                            (i) * _numberOfSlotsInboundSide + ii]);
1905                }
1906
1907                outboundscombined = new List<int>(outboundscombined.Distinct());
1908
1909                //Fill timeslots with all related outbound units
1910                for (int ii = 0;
1911                    ii < (outboundscombined.Count + _numberOfSlotsOutboundSide -
1912                          1) / _numberOfSlotsOutboundSide;
1913                    ii++)
1914                {
1915                    oneTimeSlotOutboundUnits = new List<int>();
1916                    int iii = 0;
```

```
1917                    for (iii = 0;
1918                        iii < _numberOfSlotsOutboundSide &&
1919                        (ii * _numberOfSlotsOutboundSide + iii <
1920                         outboundscombined.Count);
1921                        iii++)
1922                    {
1923                        oneTimeSlotOutboundUnits.Add(
1924                            outboundscombined[
1925                                ii * _numberOfSlotsOutboundSide + iii]);
1926                    }
1927
1928                    //If a timeslot is undefined, let the last units stay on position
1929                    for (iii = iii; iii < _numberOfSlotsOutboundSide; iii++)
1930                    {
1931                        oneTimeSlotOutboundUnits.Add(_timeSlots.Last()[1][iii]);
1932                    }
1933
1934                    _timeSlots.Add(new List<List<int>>
1935                    {
1936                        oneTimeSlotInboundUnits,
1937                        oneTimeSlotOutboundUnits
1938                    });
1939
1940
1941                    //Add previous unit if slot of timeslot is empty
1942                    if (_timeSlots.Last()[0].Count < _numberOfSlotsInboundSide)
1943                    {
1944                        for (int iiii = _timeSlots.Last()[0].Count - 1;
1945                            iiii < _numberOfSlotsInboundSide - 1;
1946                            iiii++)
1947                        {
1948                            if (_timeSlots[_timeSlots.Count - 2][0].Count ≥
1949                                iiii)
1950                            {
1951                                _timeSlots.Last()[0]
1952                                    .Add(_timeSlots[_timeSlots.Count - 2][0][
1953                                        iiii]);
1954                            }
1955                        }
1956                    }
1957
1958                    if (_timeSlots.Last()[1].Count < _numberOfSlotsOutboundSide)
1959                    {
1960                        for (int iiii = _timeSlots.Last()[1].Count - 1;
1961                            iiii < _numberOfSlotsOutboundSide - 1;
1962                            iiii++)
1963                        {
1964                            if (_timeSlots[_timeSlots.Count - 2][1].Count ≥
1965                                iiii)
1966                            {
1967                                _timeSlots.Last()[1]
1968                                    .Add(_timeSlots[_timeSlots.Count - 2][1][
1969                                        iiii]);
1970                            }
1971                        }
1972                    }
1973
```

```
1974
1975                    _linksPerTimeSlot.Add(new List<int[]>());
1976                    for (iii = 0;
1977                         iii < _numberOfSlotsInboundSide &&
1978                         (i * _numberOfSlotsInboundSide + iii <
1979                          tempInboundOutboundList.Count);
1980                         iii++)
1981                    {
1982                        List<int> linksPerInboundSlot =
1983                            tempInboundOutboundList[_timeSlots.Last()[0][iii]]
1984                                .Intersect(_timeSlots.Last()[1])
1985                                .ToList();
1986                        for (int iiii = 0;
1987                            iiii < linksPerInboundSlot.Count;
1988                            iiii++)
1989                        {
1990                            _linksPerTimeSlot.Last().Add(new int[]
1991                            {
1992                                _timeSlots.Last()[0][iii],
1993                                linksPerInboundSlot[iiii]
1994                            });
1995                        }
1996                    }
1997                }
1998            }
1999
2000            if (_approach3 == -1)
2001            {
2002                _orderedTimeSlots = _timeSlots;
2003            }
2004        }
2005
2006        private void Sim6UnitSimilarity()
2007        {
2008            //Make list to find associated outbound units for each inbound unit
2009            List<List<int>> outboundUnitsPerInboundUnit = new List<List<int>>();
2010            for (int i = 0; i < _inboundUnitList.Count; i++)
2011            {
2012                outboundUnitsPerInboundUnit.Add(_combinations
2013                    .Where(x => x[0] == i).Select(x => x[1]).ToList());
2014            }
2015
2016            List<List<int>> combinedInboundUnitsLocal = new List<List<int>>();
2017
2018            if (_numberOfSlotsInboundSide > 1)
2019            {
2020                int sharedOutboundUnits = 0;
2021
2022                List<List<int[]>> numberOfSharedOutboundUnits =
2023                    new List<List<int[]>>();
2024                for (int i = 0; i < outboundUnitsPerInboundUnit.Count - 1; i++)
2025                {
2026                    numberOfSharedOutboundUnits.Add(new List<int[]>());
2027                    for (int ii = i + 1;
2028                         ii < outboundUnitsPerInboundUnit.Count;
2029                         ii++)
2030                    {
```

```
2031                    sharedOutboundUnits = outboundUnitsPerInboundUnit[i]
2032                        .Intersect(outboundUnitsPerInboundUnit[ii])
2033                        .Count();
2034                    numberOfSharedOutboundUnits[i].Add(new int[]
2035                        {sharedOutboundUnits, i, ii});
2036                }
2037
2038                numberOfSharedOutboundUnits[i] =
2039                    numberOfSharedOutboundUnits[i]
2040                        .OrderByDescending(x => x[0]).ToList();
2041            }
2042
2043            numberOfSharedOutboundUnits = numberOfSharedOutboundUnits
2044                .OrderByDescending(x => x[0][0]).ToList();
2045
2046
2047            int slotNr = -1;
2048            while (numberOfSharedOutboundUnits.Count > 0)
2049            {
2050                //Select the best combined units
2051                numberOfSharedOutboundUnits = numberOfSharedOutboundUnits
2052                    .OrderByDescending(x => x[0][0]).ToList();
2053                combinedInboundUnitsLocal.Add(new List<int>()
2054                {
2055                    numberOfSharedOutboundUnits[0][0][1],
2056                    numberOfSharedOutboundUnits[0][0][2]
2057                });
2058                slotNr++;
2059
2060                if (_numberOfSlotsInboundSide > 2)
2061                {
2062                    List<List<int[]>> resultingAssociationLists =
2063                        new List<List<int[]>>();
2064                    //Build a list showing which unit has most outbound units
2065                    //in common with the selected units
2066                    for (int i = 0; i < 2; i++)
2067                    {
2068                        resultingAssociationLists.Add(
2069                            numberOfSharedOutboundUnits.SelectMany(x =>
2070                                x.Where(y =>
2071                                    y[1] == combinedInboundUnitsLocal[
2072                                        slotNr][i] ||
2073                                    y[2] == combinedInboundUnitsLocal[
2074                                        slotNr][i])).ToList());
2075                    }
2076
2077                    for (int i = 1; i < _numberOfSlotsInboundSide - 1; i++)
2078                    {
2079                        if (i != 1)
2080                        {
2081                            resultingAssociationLists.Add(
2082                                numberOfSharedOutboundUnits.SelectMany(x =>
2083                                    x.Where(y =>
2084                                        y[1] == combinedInboundUnitsLocal[
2085                                            slotNr][i] ||
2086                                        y[2] == combinedInboundUnitsLocal[
2087                                            slotNr][i])).ToList());
```

```
2088                        }
2089
2090                        for (int ii = 0; ii < i + 1; ii++)
2091                        {
2092                            for (int iii = 0; iii < ii; iii++)
2093                            {
2094                                resultingAssociationLists[ii].Remove(
2095                                    resultingAssociationLists[ii]
2096                                        .SingleOrDefault(x =>
2097                                            x[1] ==
2098                                            combinedInboundUnitsLocal[
2099                                                slotNr][iii] ||
2100                                            x[2] ==
2101                                            combinedInboundUnitsLocal[
2102                                                slotNr][iii]));
2103                            }
2104
2105                            for (int iii = ii + 1; iii < i + 1; iii++)
2106                            {
2107                                resultingAssociationLists[ii].Remove(
2108                                    resultingAssociationLists[ii]
2109                                        .SingleOrDefault(x =>
2110                                            x[1] ==
2111                                            combinedInboundUnitsLocal[
2112                                                slotNr][iii] ||
2113                                            x[2] ==
2114                                            combinedInboundUnitsLocal[
2115                                                slotNr][iii]));
2116                            }
2117                        }
2118
2119                        List<int> resultingPercentages = new List<int>();
2120                        resultingPercentages.AddRange(
2121                            resultingAssociationLists[0].Select(x => x[0]));
2122                        for (int ii = 1; ii < i; ii++)
2123                        {
2124                            resultingPercentages = resultingPercentages
2125                                .Zip(resultingAssociationLists[ii],
2126                                    (d1, d2) => d1 + d2[0])
2127                                .ToList();
2128                        }
2129
2130                        if (resultingPercentages.Count == 0)
2131                        {
2132                            break;
2133                        }
2134
2135                        int tempResultingList =
2136                            resultingPercentages.IndexOf(
2137                                resultingPercentages.Max());
2138                        combinedInboundUnitsLocal[slotNr]
2139                            .Add(resultingAssociationLists[0][
2140                                tempResultingList][2]);
2141                        combinedInboundUnitsLocal[slotNr]
2142                            .Add(resultingAssociationLists[0][
2143                                tempResultingList][1]);
2144                        combinedInboundUnitsLocal[slotNr] =
```

```
2145                            combinedInboundUnitsLocal[slotNr].Distinct()
2146                                .ToList();
2147                    }
2148                }
2149
2150                for (int i = 0;
2151                    i < combinedInboundUnitsLocal[slotNr].Count;
2152                    i++)
2153                {
2154                    numberOfSharedOutboundUnits.Select(x => x.RemoveAll(y =>
2155                            y[1] == combinedInboundUnitsLocal[slotNr][i] ||
2156                            y[2] == combinedInboundUnitsLocal[slotNr][i]))
2157                        .ToList();
2158                }
2159
2160                for (int i = numberOfSharedOutboundUnits.Count - 1;
2161                    i ≥ 0;
2162                    i--)
2163                {
2164                    if (numberOfSharedOutboundUnits[i].Count == 0)
2165                    {
2166                        numberOfSharedOutboundUnits.RemoveAt(i);
2167                    }
2168                }
2169            }
2170        }
2171        else
2172        {
2173            for (int i = 0; i < _inboundUnitList.Count; i++)
2174            {
2175                combinedInboundUnitsLocal.Add(new List<int> {i});
2176            }
2177        }
2178
2179        int extratimeslots = 0;
2180        List<int[]> combinationsOneTimeSlot;
2181        List<List<int[]>> combinationsPerTimeSlot = new List<List<int[]>>();
2182
2183        _timeSlots = new List<List<List<int>>>();
2184        for (int i = 0; i < combinedInboundUnitsLocal.Count; i++)
2185        {
2186            //Make new timeslot
2187            _timeSlots.Add(new List<List<int>>());
2188            _timeSlots[i + extratimeslots].Add(new List<int>());
2189            _timeSlots[i + extratimeslots].Add(new List<int>());
2190
2191            //Add the interesting outbound units for this inbound units
2192            List<int> outboundscombined = new List<int>();
2193            for (int ii = 0; ii < combinedInboundUnitsLocal[i].Count; ii++)
2194            {
2195                outboundscombined.AddRange(
2196                    outboundUnitsPerInboundUnit[
2197                        combinedInboundUnitsLocal[i][ii]]);
2198            }
2199
2200            outboundscombined = new List<int>(outboundscombined.Distinct());
2201
```

```
2202                    //For every inbound batch
2203                    bool firstTimePassing = true;
2204                    for (int ii = 0;
2205                        ii < (outboundscombined.Count + _numberOfSlotsOutboundSide -
2206                              1) / _numberOfSlotsOutboundSide;
2207                        ii++)
2208                    {
2209                        //Add new timeslot if it is not the first time
2210                        if (!firstTimePassing)
2211                        {
2212                            extratimeslots++;
2213                            _timeSlots.Add(new List<List<int>>());
2214                            _timeSlots[i + extratimeslots].Add(new List<int>());
2215                            _timeSlots[i + extratimeslots].Add(new List<int>());
2216                        }
2217
2218                        firstTimePassing = false;
2219                        combinationsOneTimeSlot = new List<int[]>();
2220
2221                        for (int iii = 0;
2222                            iii < combinedInboundUnitsLocal[i].Count;
2223                            iii++)
2224                        {
2225                            _timeSlots[i + extratimeslots][0]
2226                                .Add(combinedInboundUnitsLocal[i][iii]);
2227                        }
2228
2229                        for (int iii = 0; iii < _numberOfSlotsOutboundSide; iii++)
2230                        {
2231                            if (outboundscombined.Count >
2232                                ii * _numberOfSlotsOutboundSide + iii)
2233                            {
2234                                _timeSlots[i + extratimeslots][1]
2235                                    .Add(outboundscombined[
2236                                        ii * _numberOfSlotsOutboundSide + iii]);
2237                                var result = _combinations.Where(x =>
2238                                        combinedInboundUnitsLocal[i]
2239                                            .Any(y => y == x[0]) &&
2240                                        x[1] == outboundscombined[
2241                                            ii * _numberOfSlotsOutboundSide + iii])
2242                                    .ToList();
2243                                combinationsOneTimeSlot.AddRange(result);
2244                                _combinations.RemoveAll(x =>
2245                                    combinedInboundUnitsLocal[i]
2246                                        .Any(y => y == x[0]) &&
2247                                    x[1] == outboundscombined[
2248                                        ii * _numberOfSlotsOutboundSide + iii]);
2249                            }
2250                        }
2251
2252                        combinationsPerTimeSlot.Add(combinationsOneTimeSlot);
2253
2254                        //Add previous unit if slot of timeslot is empty
2255                        if (_timeSlots.Last()[0].Count < _numberOfSlotsInboundSide)
2256                        {
2257                            for (int iii = _timeSlots.Last()[0].Count - 1;
2258                                iii < _numberOfSlotsInboundSide - 1;
```

```
2259                            iii++)
2260                        {
2261                            if (_timeSlots[_timeSlots.Count - 2][0].Count ≥
2262                                iii)
2263                            {
2264                                _timeSlots.Last()[0]
2265                                    .Add(_timeSlots[_timeSlots.Count - 2][0][
2266                                        iii]);
2267                            }
2268                        }
2269                    }
2270
2271                    if (_timeSlots.Last()[1].Count < _numberOfSlotsOutboundSide)
2272                    {
2273                        for (int iii = _timeSlots.Last()[1].Count - 1;
2274                            iii < _numberOfSlotsOutboundSide - 1;
2275                            iii++)
2276                        {
2277                            if (_timeSlots[_timeSlots.Count - 2][1].Count ≥
2278                                iii)
2279                            {
2280                                _timeSlots.Last()[1]
2281                                    .Add(_timeSlots[_timeSlots.Count - 2][1][
2282                                        iii]);
2283                            }
2284                        }
2285                    }
2286                }
2287            }
2288        }
2289
2290        private void Sim7DynamicChanging()
2291        {
2292            List<List<int>> linksForThisTimeSlotInbound = new List<List<int>>();
2293            List<List<int>> linksForThisTimeSlotOutbound =
2294                new List<List<int>>();
2295            List<int> inboundUnitsForAllOutboundUnitsAtTimeSlot =
2296                new List<int>();
2297            List<int> outboundUnitsForAllInboundUnitsAtTimeSlot =
2298                new List<int>();
2299
2300            //Make list to find associated outbound units for each inbound unit
2301            List<List<int>> outboundUnitsPerInboundUnit = new List<List<int>>();
2302            for (int i = 0; i < _inboundUnitList.Count; i++)
2303            {
2304                outboundUnitsPerInboundUnit.Add(_combinations
2305                    .Where(x => x[0] == i).Select(x => x[1]).ToList());
2306            }
2307
2308            //Make list to find shared inbound units for each outbound unit
2309            List<List<int>> inboundUnitsPerOutboundUnit = new List<List<int>>();
2310            for (int i = 0; i < _outboundUnitList.Count; i++)
2311            {
2312                inboundUnitsPerOutboundUnit.Add(_combinations
2313                    .Where(x => x[1] == i).Select(x => x[0]).ToList());
2314            }
2315
```

```
2316            //Add first units to inbound side
2317            List<List<int>> timeSlot =
2318                new List<List<int>> {new List<int>(), new List<int>()};
2319            for (int i = 0; i < _numberOfSlotsInboundSide; i++)
2320            {
2321                timeSlot[0].Add(i);
2322            }
2323
2324            List<int[]> numberOfSharedOutboundUnitsInboundSide =
2325                new List<int[]>();
2326            for (int i = 0; i < _outboundUnitList.Count; i++)
2327            {
2328                numberOfSharedOutboundUnitsInboundSide.Add(new int[]
2329                {
2330                    timeSlot[0].Intersect(inboundUnitsPerOutboundUnit[i])
2331                        .Count(),
2332                    i
2333                });
2334            }
2335
2336            numberOfSharedOutboundUnitsInboundSide =
2337                numberOfSharedOutboundUnitsInboundSide
2338                    .OrderByDescending(x => x[0]).ToList();
2339
2340            //Add outbound units with most links with the inbound units
2341            for (int i = 0; i < _numberOfSlotsOutboundSide; i++)
2342            {
2343                timeSlot[1].Add(numberOfSharedOutboundUnitsInboundSide[i][1]);
2344            }
2345
2346            List<List<int>> timeSlotTemp =
2347                timeSlot.ConvertAll(x => new List<int>(x));
2348            _timeSlots.Add(timeSlotTemp);
2349            _linksPerTimeSlot.Add(new List<int[]>());
2350            int pos = 0;
2351
2352            while (outboundUnitsPerInboundUnit.Sum(x => x.Count) > 0)
2353            {
2354                //Get lists of the possible links
2355                linksForThisTimeSlotInbound = new List<List<int>>();
2356                linksForThisTimeSlotOutbound = new List<List<int>>();
2357
2358                //Find the matches for each in and outbound unit at this timeslot
2359                for (int i = 0; i < _numberOfSlotsOutboundSide; i++)
2360                {
2361                    linksForThisTimeSlotOutbound.Add(
2362                        inboundUnitsPerOutboundUnit[timeSlot[1][i]]
2363                            .Intersect(timeSlot[0])
2364                            .ToList());
2365                }
2366
2367                for (int i = 0; i < _numberOfSlotsInboundSide; i++)
2368                {
2369                    linksForThisTimeSlotInbound.Add(new List<int>());
2370                }
2371
2372                for (int i = 0; i < _numberOfSlotsOutboundSide; i++)
```

```
2373                {
2374                    for (int ii = 0;
2375                        ii < linksForThisTimeSlotOutbound[i].Count;
2376                        ii++)
2377                    {
2378                        linksForThisTimeSlotInbound[
2379                                timeSlot[0]
2380                                    .IndexOf(
2381                                        linksForThisTimeSlotOutbound[i][ii])]
2382                            .Add(timeSlot[1][i]);
2383                    }
2384                }
2385
2386                //If there are no more links for this timeslot. Replace the
2387                //inbound side for the first units having links with other units.
2388                if (linksForThisTimeSlotInbound.Sum(x => x.Count) == 0)
2389                {
2390                    int ii = 0;
2391                    for (int i = 0; i < _numberOfSlotsInboundSide; i++)
2392                    {
2393                        for (ii = ii;
2394                            ii < outboundUnitsPerInboundUnit.Count;
2395                            ii++)
2396                        {
2397                            if (outboundUnitsPerInboundUnit[ii].Count > 0)
2398                            {
2399                                timeSlot[0][i] = ii;
2400                                ii++;
2401                                break;
2402                            }
2403                        }
2404                    }
2405
2406                    //And replace the outbound side with the units having
2407                    //most matches with the inbound side.
2408                    for (int i = 0; i < _numberOfSlotsInboundSide; i++)
2409                    {
2410                        outboundUnitsForAllInboundUnitsAtTimeSlot.AddRange(
2411                            outboundUnitsPerInboundUnit[timeSlot[0][i]]);
2412                    }
2413
2414                    for (int i = 0; i < _numberOfSlotsOutboundSide; i++)
2415                    {
2416                        if (linksForThisTimeSlotOutbound[i].Count == 0)
2417                        {
2418                            outboundUnitsForAllInboundUnitsAtTimeSlot
2419                                .GroupBy(x => x).MaxBy(y => y.Count());
2420                            outboundUnitsForAllInboundUnitsAtTimeSlot =
2421                                outboundUnitsForAllInboundUnitsAtTimeSlot
2422                                    .Except(timeSlot[1]).ToList();
2423
2424                            if (outboundUnitsForAllInboundUnitsAtTimeSlot
2425                                    .Count > 0)
2426                            {
2427                                timeSlot[1][i] =
2428                                    outboundUnitsForAllInboundUnitsAtTimeSlot
2429                                        [0];
```

```
2430                        }
2431                    }
2432                }
2433            }
2434            else
2435            {
2436                if (!timeSlot[0].All(_timeSlots.Last()[0].Contains) ||
2437                    !timeSlot[1].All(_timeSlots.Last()[1].Contains))
2438                {
2439                    timeSlotTemp = new List<List<int>>();
2440                    timeSlotTemp =
2441                        timeSlot.ConvertAll(x => new List<int>(x));
2442                    _timeSlots.Add(timeSlotTemp);
2443                    _linksPerTimeSlot.Add(new List<int[]>());
2444                }
2445
2446                int tempCount = 0;
2447                while (linksForThisTimeSlotInbound[pos].Count <= 0)
2448                {
2449                    pos++;
2450                    if (pos >= _numberOfSlotsInboundSide)
2451                    {
2452                        pos = 0;
2453                    }
2454
2455                    tempCount++;
2456                    if (tempCount > _numberOfSlotsInboundSide)
2457                    {
2458                        break;
2459                    }
2460                }
2461
2462                _linksPerTimeSlot.Last().Add(new int[]
2463                {
2464                    timeSlot[0][pos], linksForThisTimeSlotInbound[pos][0]
2465                });
2466                inboundUnitsPerOutboundUnit[
2467                        linksForThisTimeSlotInbound[pos][0]]
2468                    .Remove(timeSlot[0][pos]);
2469                outboundUnitsPerInboundUnit[timeSlot[0][pos]]
2470                    .Remove(linksForThisTimeSlotInbound[pos][0]);
2471                linksForThisTimeSlotOutbound[
2472                        timeSlot[1]
2473                            .IndexOf(linksForThisTimeSlotInbound[pos][0])]
2474                    .Remove(timeSlot[0][pos]);
2475                linksForThisTimeSlotInbound[pos].RemoveAt(0);
2476            }
2477
2478
2479            //Find the best unit to replace an empty one (start with inbound side)
2480            outboundUnitsForAllInboundUnitsAtTimeSlot = new List<int>();
2481            for (int i = 0; i < _numberOfSlotsInboundSide; i++)
2482            {
2483                outboundUnitsForAllInboundUnitsAtTimeSlot.AddRange(
2484                    outboundUnitsPerInboundUnit[timeSlot[0][i]]);
2485            }
2486
```

```
2487                for (int i = 0; i < _numberOfSlotsOutboundSide; i++)
2488                {
2489                    if (linksForThisTimeSlotOutbound[i].Count == 0)
2490                    {
2491                        var kaas = outboundUnitsForAllInboundUnitsAtTimeSlot
2492                            .GroupBy(x => x)
2493                            .OrderByDescending(y => y.Count()).ToList();
2494                        List<int> kaas2 = new List<int>();
2495                        for (int ii = 0; ii < kaas.Count(); ii++)
2496                        {
2497                            kaas2.Add(kaas[ii].Key);
2498                        }
2499
2500                        outboundUnitsForAllInboundUnitsAtTimeSlot =
2501                            kaas2.Except(timeSlot[1]).ToList();
2502
2503                        if (outboundUnitsForAllInboundUnitsAtTimeSlot.Count > 0)
2504                        {
2505                            timeSlot[1][i] =
2506                                outboundUnitsForAllInboundUnitsAtTimeSlot[0];
2507                        }
2508                    }
2509                }
2510
2511
2512                inboundUnitsForAllOutboundUnitsAtTimeSlot = new List<int>();
2513                for (int i = 0; i < _numberOfSlotsOutboundSide; i++)
2514                {
2515                    inboundUnitsForAllOutboundUnitsAtTimeSlot.AddRange(
2516                        inboundUnitsPerOutboundUnit[timeSlot[1][i]]);
2517                }
2518
2519                for (int i = 0; i < _numberOfSlotsInboundSide; i++)
2520                {
2521                    if (linksForThisTimeSlotInbound[i].Count == 0)
2522                    {
2523                        var kaas = inboundUnitsForAllOutboundUnitsAtTimeSlot
2524                            .GroupBy(x => x)
2525                            .OrderByDescending(y => y.Count()).ToList();
2526                        List<int> kaas2 = new List<int>();
2527                        for (int ii = 0; ii < kaas.Count(); ii++)
2528                        {
2529                            kaas2.Add(kaas[ii].Key);
2530                        }
2531
2532                        inboundUnitsForAllOutboundUnitsAtTimeSlot =
2533                            kaas2.Except(timeSlot[0]).ToList();
2534
2535                        if (inboundUnitsForAllOutboundUnitsAtTimeSlot.Count > 0)
2536                        {
2537                            timeSlot[0][i] =
2538                                inboundUnitsForAllOutboundUnitsAtTimeSlot[0];
2539                        }
2540                    }
2541                }
2542            }
2543
```

```
2544            _orderedTimeSlots = _timeSlots;
2545        }
2546
2547        private void Sim8DynamicBatching()
2548        {
2549            List<int> linksPerOutboundSlot = new List<int>();
2550
2551            //Make list to find associated outbound units for each inbound unit
2552            List<List<int>> outboundUnitsPerInboundUnit = new List<List<int>>();
2553            for (int i = 0; i < _inboundUnitList.Count; i++)
2554            {
2555                outboundUnitsPerInboundUnit.Add(_combinations
2556                    .Where(x => x[0] == i).Select(x => x[1]).ToList());
2557            }
2558
2559            //Make list to find shared inbound units for each outbound unit
2560            List<List<int>> inboundUnitsPerOutboundUnit = new List<List<int>>();
2561            for (int i = 0; i < _outboundUnitList.Count; i++)
2562            {
2563                inboundUnitsPerOutboundUnit.Add(_combinations
2564                    .Where(x => x[1] == i).Select(x => x[0]).ToList());
2565            }
2566
2567            //Add first units to inbound side
2568            List<List<int>> timeSlot =
2569                new List<List<int>> {new List<int>(), new List<int>()};
2570            for (int i = 0; i < _numberOfSlotsInboundSide; i++)
2571            {
2572                timeSlot[0].Add(i);
2573            }
2574
2575            List<int[]> numberOfSharedOutboundUnitsInboundSide =
2576                new List<int[]>();
2577            for (int i = 0; i < _outboundUnitList.Count; i++)
2578            {
2579                numberOfSharedOutboundUnitsInboundSide.Add(new int[]
2580                {
2581                    timeSlot[0].Intersect(inboundUnitsPerOutboundUnit[i])
2582                        .Count(),
2583                    i
2584                });
2585            }
2586
2587            numberOfSharedOutboundUnitsInboundSide =
2588                numberOfSharedOutboundUnitsInboundSide
2589                    .OrderByDescending(x => x[0]).ToList();
2590
2591            //Add outbound units with most links with the inbound units
2592            for (int i = 0; i < _numberOfSlotsOutboundSide; i++)
2593            {
2594                timeSlot[1].Add(numberOfSharedOutboundUnitsInboundSide[i][1]);
2595            }
2596
2597            List<List<int>> timeSlotTemp =
2598                timeSlot.ConvertAll(x => new List<int>(x));
2599            _timeSlots.Add(timeSlotTemp);
2600            _linksPerTimeSlot.Add(new List<int[]>());
```

```
2601
2602            while (outboundUnitsPerInboundUnit.Sum(x => x.Count) > 0)
2603            {
2604                int test = outboundUnitsPerInboundUnit.Sum(x => x.Count);
2605                //Find the matches for each in and outbound unit at this timeslot
2606                for (int i = 0; i < _numberOfSlotsOutboundSide; i++)
2607                {
2608                    //Find all matching indices per timeslot and remove them
2609                    linksPerOutboundSlot =
2610                        new List<int>(
2611                            inboundUnitsPerOutboundUnit[timeSlot[1][i]]
2612                                .Intersect(timeSlot[0]).ToList());
2613                    for (int ii = 0; ii < linksPerOutboundSlot.Count; ii++)
2614                    {
2615                        _linksPerTimeSlot.Last().Add(new int[]
2616                            {linksPerOutboundSlot[ii], timeSlot[1][i]});
2617                        inboundUnitsPerOutboundUnit[timeSlot[1][i]]
2618                            .Remove(linksPerOutboundSlot[ii]);
2619                        outboundUnitsPerInboundUnit[linksPerOutboundSlot[ii]]
2620                            .Remove(timeSlot[1][i]);
2621                    }
2622                }
2623
2624                //Find the units with most links to the outbound timeslot
2625                List<int[]> numberOfSharedOutboundUnitsPerInboundSlot =
2626                    new List<int[]>();
2627                for (int i = 0; i < _outboundUnitList.Count; i++)
2628                {
2629                    if (timeSlot[0].Intersect(inboundUnitsPerOutboundUnit[i])
2630                            .Count() > 0)
2631                    {
2632                        numberOfSharedOutboundUnitsPerInboundSlot.Add(new int[]
2633                        {
2634                            timeSlot[0]
2635                                .Intersect(inboundUnitsPerOutboundUnit[i])
2636                                .Count(),
2637                            i
2638                        });
2639                    }
2640                }
2641
2642                numberOfSharedOutboundUnitsPerInboundSlot =
2643                    numberOfSharedOutboundUnitsPerInboundSlot
2644                        .OrderByDescending(x => x[0]).ToList();
2645
2646                int changeOutboundSide = 0;
2647
2648                for (int i = 0;
2649                    i < _numberOfSlotsOutboundSide && i <
2650                    numberOfSharedOutboundUnitsPerInboundSlot.Count;
2651                    i++)
2652                {
2653                    changeOutboundSide +=
2654                        numberOfSharedOutboundUnitsPerInboundSlot[i][0];
2655                }
2656
2657                //Find the units with most links to the inbound timeslot
```

```
2658                List<int[]> numberOfSharedInboundUnitsPerOutboundSlot =
2659                    new List<int[]>();
2660                for (int i = 0; i < _inboundUnitList.Count; i++)
2661                {
2662                    if (timeSlot[1].Intersect(outboundUnitsPerInboundUnit[i])
2663                            .Count() > 0)
2664                    {
2665                        numberOfSharedInboundUnitsPerOutboundSlot.Add(new int[]
2666                        {
2667                            timeSlot[1]
2668                                .Intersect(outboundUnitsPerInboundUnit[i])
2669                                .Count(),
2670                            i
2671                        });
2672                    }
2673                }
2674
2675                numberOfSharedInboundUnitsPerOutboundSlot =
2676                    numberOfSharedInboundUnitsPerOutboundSlot
2677                        .OrderByDescending(x => x[0]).ToList();
2678
2679                int changeInboundSide = 0;
2680
2681                for (int i = 0;
2682                    i < _numberOfSlotsInboundSide && i <
2683                    numberOfSharedInboundUnitsPerOutboundSlot.Count;
2684                    i++)
2685                {
2686                    changeInboundSide +=
2687                        numberOfSharedInboundUnitsPerOutboundSlot[i][0];
2688                }
2689
2690                if (changeInboundSide ≥ changeOutboundSide)
2691                {
2692                    for (int i = 0;
2693                        i < _numberOfSlotsInboundSide && i <
2694                        numberOfSharedInboundUnitsPerOutboundSlot.Count;
2695                        i++)
2696                    {
2697                        timeSlot[0][i] =
2698                            numberOfSharedInboundUnitsPerOutboundSlot[i][1];
2699                    }
2700
2701                    int ii = 0;
2702                    for (int
2703                        i = numberOfSharedInboundUnitsPerOutboundSlot.Count;
2704                        i < _numberOfSlotsInboundSide;
2705                        i++)
2706                    {
2707                        for (ii = ii;
2708                            ii < outboundUnitsPerInboundUnit.Count;
2709                            ii++)
2710                        {
2711                            if (outboundUnitsPerInboundUnit[ii].Count > 0 &&
2712                                !timeSlot[0].GetRange(0, i).Contains(ii))
2713                            {
2714                                timeSlot[0][i] = ii;
```

```
2715                                    break;
2716                                }
2717                            }
2718                        }
2719                    }
2720                    else
2721                    {
2722                        for (int i = 0;
2723                            i < _numberOfSlotsOutboundSide && i <
2724                            numberOfSharedOutboundUnitsPerInboundSlot.Count;
2725                            i++)
2726                        {
2727                            timeSlot[1][i] =
2728                                numberOfSharedOutboundUnitsPerInboundSlot[i][1];
2729                        }
2730
2731                        int ii = 0;
2732                        for (int
2733                            i = numberOfSharedOutboundUnitsPerInboundSlot.Count;
2734                            i < _numberOfSlotsOutboundSide;
2735                            i++)
2736                        {
2737                            for (ii = ii;
2738                                ii < inboundUnitsPerOutboundUnit.Count;
2739                                ii++)
2740                            {
2741                                if (inboundUnitsPerOutboundUnit[ii].Count > 0 &&
2742                                    !timeSlot[1].GetRange(0, i).Contains(ii))
2743                                {
2744                                    timeSlot[1][i] = ii;
2745                                    break;
2746                                }
2747                            }
2748                        }
2749                    }
2750
2751                    timeSlotTemp = new List<List<int>>();
2752                    timeSlotTemp = timeSlot.ConvertAll(x => new List<int>(x));
2753                    _timeSlots.Add(timeSlotTemp);
2754                    _linksPerTimeSlot.Add(new List<int[]>());
2755                }
2756
2757                _orderedTimeSlots = _timeSlots;
2758            }
2759
2760            private void Sim9MinimumChanges()
2761            {
2762                //Make a list with the number of similar units between te timeslots.
2763                _orderedTimeSlots = new List<List<List<int>>>();
2764                int numberOfIntersects = 0;
2765                List<int> orderedTimeSlotsIndices = new List<int>();
2766                List<List<int[]>> batchingCombinations = new List<List<int[]>>();
2767                for (int i = 0; i < _timeSlots.Count; i++)
2768                {
2769                    batchingCombinations.Add(new List<int[]>());
2770                    for (int ii = i + 1; ii < _timeSlots.Count; ii++)
2771                    {
```

```
2772                    numberOfIntersects = _timeSlots[i][1]
2773                        .Intersect(_timeSlots[ii][1]).Count();
2774                    if (numberOfIntersects > 0)
2775                    {
2776                        batchingCombinations[i].Add(new int[]
2777                            {numberOfIntersects, i, ii});
2778                    }
2779                }
2780
2781                batchingCombinations[i] = batchingCombinations[i]
2782                    .OrderByDescending(x => x[0]).ToList();
2783            }
2784
2785            batchingCombinations.RemoveAll(x => x.Count == 0);
2786            batchingCombinations.OrderByDescending(x => x[0][0]).ToList();
2787
2788            var heighestNumber = batchingCombinations[0][0][0];
2789            var heighestIndex = batchingCombinations[0][0][1];
2790
2791            //The found combination are two sequential time slots.
2792            List<int> definedCombinationsOrder = new List<int>();
2793            definedCombinationsOrder.Add(
2794                batchingCombinations[heighestIndex][0][1]);
2795            definedCombinationsOrder.Add(
2796                batchingCombinations[heighestIndex][0][2]);
2797            var linkedSlots2 =
2798                GetTimeSlotsWithTheSameInboundUnitsAs(
2799                    definedCombinationsOrder[0]);
2800            orderedTimeSlotsIndices.AddRange(linkedSlots2);
2801            orderedTimeSlotsIndices.AddRange(definedCombinationsOrder);
2802            batchingCombinations.ForEach(x => x.RemoveAll(y =>
2803                linkedSlots2.Any(z => z == y[1] || z == y[2]) ||
2804                y[1] == definedCombinationsOrder[0] ||
2805                y[2] == definedCombinationsOrder[0]));
2806            batchingCombinations.RemoveAll(x => x.Count == 0);
2807
2808            while (orderedTimeSlotsIndices.Count < _timeSlots.Count)
2809            {
2810                List<int> linkedSlots =
2811                    GetTimeSlotsWithTheSameInboundUnitsAs(
2812                        definedCombinationsOrder.Last());
2813                List<int[]> nextBest = new List<int[]>();
2814                double time = DateTime.Now.Ticks;
2815
2816                List<int> timeSlotsLeft = new List<int>();
2817                for (int i = 0; i < _timeSlots.Count; i++)
2818                {
2819                    timeSlotsLeft.Add(i);
2820                }
2821
2822                Stopwatch stopwatch = new Stopwatch();
2823                stopwatch.Start();
2824                nextBest = batchingCombinations
2825                    .SelectMany(x => x.Where(y =>
2826                        (linkedSlots.Any(z => z == y[1] || z == y[2]))))
2827                    .ToList();
2828                nextBest = nextBest
```

```
2829                .Where(x => (linkedSlots.Any(y => y == x[1])
2830                             && !linkedSlots.Any(z => z == x[2])
2831                             && x[2] != definedCombinationsOrder.Last()
2832                             ||
2833                             linkedSlots.Any(y => y == x[2])
2834                             && !linkedSlots.Any(z => z == x[1])
2835                             && x[1] != definedCombinationsOrder.Last()))
2836                .MaxBy(x => x[0]).ToList();
2837
2838            if (nextBest.Count == 0)
2839            {
2840                nextBest = batchingCombinations.SelectMany(x => x.Where(y =>
2841                        (y[1] == definedCombinationsOrder.Last() &&
2842                         y[2] != definedCombinationsOrder[
2843                             definedCombinationsOrder.Count - 2] ||
2844                         y[2] == definedCombinationsOrder.Last() &&
2845                         y[1] != definedCombinationsOrder[
2846                             definedCombinationsOrder.Count - 2])))
2847                    .MaxBy(x => x[0])
2848                    .ToList();
2849            }
2850
2851            linkedSlots.Add(definedCombinationsOrder.Last());
2852
2853            batchingCombinations.ForEach(x =>
2854                x.RemoveAll(y =>
2855                    linkedSlots.Any(z => z == y[1] || z == y[2])));
2856
2857            batchingCombinations.RemoveAll(x => x.Count == 0);
2858
2859            if (nextBest.Count == 0)
2860            {
2861                timeSlotsLeft = timeSlotsLeft
2862                    .Except(orderedTimeSlotsIndices).ToList();
2863                nextBest.Add(new int[]
2864                    {0, definedCombinationsOrder.Last(), timeSlotsLeft[0]});
2865            }
2866
2867            if (linkedSlots.Any(x => x == nextBest[0][1]))
2868            {
2869                if (nextBest[0][1] != definedCombinationsOrder.Last())
2870                {
2871                    definedCombinationsOrder.Add(nextBest[0][1]);
2872                }
2873
2874                definedCombinationsOrder.Add(nextBest[0][2]);
2875            }
2876            else
2877            {
2878                definedCombinationsOrder.Add(nextBest[0][2]);
2879                definedCombinationsOrder.Add(nextBest[0][1]);
2880            }
2881
2882            linkedSlots = linkedSlots.Where(x =>
2883                x != definedCombinationsOrder[
2884                    definedCombinationsOrder.Count - 2] &&
2885                x != definedCombinationsOrder[
```

```
2886                        definedCombinationsOrder.Count - 3]).ToList();
2887                if (orderedTimeSlotsIndices.Last() !=
2888                    definedCombinationsOrder[definedCombinationsOrder.Count - 2]
2889                )
2890                {
2891                    orderedTimeSlotsIndices.Add(
2892                        definedCombinationsOrder[
2893                            definedCombinationsOrder.Count - 2]);
2894                }
2895
2896                orderedTimeSlotsIndices.Add(
2897                    definedCombinationsOrder
2898                        [definedCombinationsOrder.Count - 1]);
2899                orderedTimeSlotsIndices.InsertRange(
2900                    orderedTimeSlotsIndices.Count - 2, linkedSlots);
2901            }
2902
2903            _orderedTimeSlots = GetTimeSlotsByIndex(orderedTimeSlotsIndices);
2904        }
2905
2906
2907        private List<List<List<int>>> GetTimeSlotsByIndex(
2908            List<int> timeSlotIndices)
2909        {
2910            List<List<List<int>>> obtainedTimeSlots =
2911                new List<List<List<int>>>();
2912            for (int i = 0; i < timeSlotIndices.Count - 1; i++)
2913            {
2914                obtainedTimeSlots.Add(_timeSlots[timeSlotIndices[i]]);
2915            }
2916
2917            return obtainedTimeSlots;
2918        }
2919
2920        private List<int> GetTimeSlotsWithTheSameInboundUnitsAs(int timeSlotNr)
2921        {
2922            List<int> timeSlotsWithSameInboundUnits = new List<int>();
2923            int t = timeSlotNr + 1;
2924            while (t < _timeSlots.Count && _timeSlots[t][0]
2925                       .Except(_timeSlots[timeSlotNr][0]).Count() == 0)
2926            {
2927                timeSlotsWithSameInboundUnits.Add(t);
2928                t++;
2929            }
2930
2931            t = timeSlotNr - 1;
2932            while (t ≥ 0 && _timeSlots[t][0].Except(_timeSlots[timeSlotNr][0])
2933                       .Count() == 0)
2934            {
2935                timeSlotsWithSameInboundUnits.Add(t);
2936                t--;
2937            }
2938
2939            return timeSlotsWithSameInboundUnits;
2940        }
2941
2942        private List<int[]> CreateSubSetList(int element,
```

```
2943                List<List<OrderLine>> inOrOutUnitList)
2944            {
2945                List<int[]> subSetList = new List<int[]>();
2946                for (int i = 0; i < inOrOutUnitList.Count; i++)
2947                {
2948                    List<OrderLine> tempList = inOrOutUnitList[i]
2949                        .Where(x => x.ArticleNumberCustomer == element).ToList();
2950                    if (tempList.Count > 0)
2951                    {
2952                        int nrOfProducts = 0;
2953                        foreach (var tempElement in tempList)
2954                        {
2955                            nrOfProducts += tempElement.NumberOfProductsUnused;
2956                        }
2957
2958                        subSetList.Add(new int[] {nrOfProducts, i});
2959                    }
2960                }
2961
2962                return subSetList;
2963            }
2964        }
2965    }
```