LUND UNIVERSITY

LTH

FACULTY OF ENGINEERING

MASTER'S THESIS
MIOM05 Degree Project in Production Management

Consolidation During Picking

Faculty of Engineering LTH
Department of Mechanical Engineering Sciences
Division of Production Management

Author: Karl Magnus Rosell
Supervisors: Johan Marklund, Faculty of Engineering LTH
& Erik Berggren, Industri-Matematik International AB

March 2026

Abstract

Title: Consolidation During Picking
Author: Karl Magnus Rosell
Supervisors: Johan Marklund, Lund University | Erik Berggren, Industri-Matematik International AB
Background: Order-picking accounts for more than half of total warehouse operating costs, where travel time is the largest contributor. At the case company, a Swedish healthcare consumables distributor served by Industri-Matematik International AB, a cluster-picking strategy is implemented to improve warehouse picking efficiency. This strategy is combined with a routing approach based on the combined routing method proposed by Roodbergen (2001). The current picking process lacks planning of how tote boxes (orders) should be allocated to unit-loads such as pallets, resulting in unnecessary additions of boxes onto unit-loads as well as extra picking rounds due to the precedence constrains imposed by the S-shaped routing strategy.
Purpose: The purpose of the thesis is to develop a new pick planning algorithm for the case company that reduces the total time in the picking phase by optimizing the picking route and box allocations on the unit-load.
Methodology: The methodology was based on a six step operations research framework. First, warehouse and historical pick-order data were collected and mapped to an abstract model of the warehouse and routing policy. Thereafter, a recursive depth-first search pick-path optimization algorithm was developed for finding the shortest route, using the algorithm described by Bartholdi & Hackman (2019, p. 163). Thereafter, three search algorithms — brute-force, random search and branch-and-bound — were implemented and compared for generating unit-load configurations.
Conclusions: The developed algorithm prevents the addition of unnecessary boxes during picking and all the search algorithms outperformed the serpentine baseline routing across the tested cases. For actual pick-order cases, the serpentine baseline was approximately 245% to 325% longer compared to the computed picking routes, while for generated pick-orders involving higher number of orders, the serpentine baseline was approximately 130% to 170% longer compared to the computed picking routes. Branch-and-bound is recommended for pick-orders with up to 12-16 orders depending on the number of orders per layer, while random search is recommended for the larger pick-orders. For pick-orders where the unique-order stacks are sufficient to fill complete layers, a combined allocation heuristic of random search and order-stack prioritization is recommended. In conclusion, the algorithm has the potential to improve manual handling and picking efficiency if certain constraints are relaxed, and can serve as a foundation for further optimization of box allocation.
Keywords: Pick-path optimization, Order-picking, Serpentine routing, Precedence constraints, Heuristic search

Preface

This thesis was conducted during the winter of 2025 and spring of 2026 as part of the Mechanical Engineering program at Faculty of Engineering LTH, specializing in logistics and production management.

I would like to thank my thesis supervisor, Johan Marklund, for his valuable time and support. His insights were crucial in guiding me in the design of the main solution framework. I am also grateful for the opportunity to conduct my thesis at IMI, and I want to thank my supervisor, Erik Berggren, for his guidance and understanding throughout the project. His perception of the problem and advice on potential solutions during discussions were invaluable. Lastly, I want to thank Filip Månsson and Göran Ingvarsson for their continuous encouragement and support.

- Karl Magnus Rosell, March 2026, Lund

Contents

List of Figures

List of Tables

Abbreviations

Notation	Description
IMI	Industri-Matematik International
WMS	Warehouse Management System
SKU	Stock Keeping Unit
ULC	Unit-Load Configuration
TSP	Traveling Salesman Problem
OBP	Order-Batching Problem
OS	Order-Stack
UOS	Unique Order-Stack
PLC	Pick-Load Carrier
POL	Pick-Order Line
GUI	Graphical User Interface
BF	Brute-force
B&B	Branch-and-Bound
RS	Random search

Table 1: Abbreviations

Table of definitions

Term	Physical equivalent
Pick-order	Complete set of customer orders to fulfill in one picking session.
Unit-load	A material handling device (e.g., Euro-pallet, roll container) onto which tote boxes are stacked during picking.
Unit-load configuration (ULC)	The arrangement of boxes across layers on a unit-load, derived from a pick-order.
Pick-load carrier (PLC)	A tote box destined for a single customer order, placed on the unit-load.
Pick-order line (POL)	Specifies item to pick to a PLC as well as its quantity and location
Picking round	One complete traversal of the pick path to collect items for a set of boxes.
Order-stack (OS)	A pairing of two orders/tote boxes where the first is completed before the second begins, allowing them to share a unit-load position across layers.
Unique order-stack (UOS)	An order-stack in which each order/tote box appears in no other order-stack, ensuring that involved tote boxes only appear once on the unit-load.
Box accessibility span	The span during which a box is accessible for picks, determined by the position of its first and last item pick in the serpentine path.

Table 2: Term definitions and their physical equivalent.

1. Introduction

In this section, the background and context of the thesis is described. Also, the case company, problem description, purpose and limitations of the thesis are presented.

1.1 Background

Order-picking is widely recognized as the most labor-intensive and cost-driving activity in warehouse operations and usually accounts for more than half of total operating expenses according to De Koster, et al. (2007). A large portion of this cost is due to manual handling, but also to travel time, which represents more than 50% of the total item retrieval time (Tompkins et al., 2010). To reduce these warehouse outbound costs, cluster-picking can prove effective for certain picker-to-parts warehouses as it merges picking with the packing process. In cluster-picking, multiple customer orders are picked simultaneously in a picking round and consolidated into unit-loads (e.g Euro-pallets) to reduce travel distances and improve productivity.

In this thesis, precedence constraints dictate the stacking principles of tote boxes (representing customer orders) on unit-loads based on the pick sequence. The constraint is imposed as a serpentine routing strategy determines the pick sequence. The stacking principles require that open tote boxes with remaining picks further in the path should not be covered when introducing other tote boxes to the next layer on the unit-load. This constraint combined with the limited routing possibilities that the serpentine-path offers, leads to increased complexity in planning the pick sequence and box allocations onto the unit-load.

1.1.1 Warehouse management system and cluster-picking

IMI currently maintains one of their products, a Warehouse Management System (WMS) for one of their customers, a large healthcare consumables logistics provider in Sweden. The WMS is used for managing the inventory handling activities as stated by Bartholdi and Hackman (2019, p. 36). The WMS systems also typically integrate with the broader ERP-system, where access to financial data allows the WMS to achieve greater supply chain visibility and enhanced functionality in logistics. For instance, functions such as *order fulfillment* require a connection to invoicing, and inventory updates require data from purchasing and sales. Upon receiving a customer order,

the WMS reviews inventory availability, generates pick lists, and coordinates outbound activities to ensure accurate and timely deliveries.

In the WMS provided by IMI, cluster-picking is supported and often used in customer fulfillment centers. The pick-order instructs the order-picker which items to retrieve, where they are located in the warehouse, and in which tote box, also referred to as a pick-load carrier (PLC), to place them. Items, also referred to as Stock Keeping Units (SKUs) are picked into separate boxes for multiple orders in the same picking round. Cluster-picking is often used in practice when the order fulfillment process is characterized by a high amount of orders with relatively few pick-order lines (POL) per order and where the orders frequently require the same high-demand articles. By adopting a cluster-picking strategy, the order-pickers avoid repeatedly visiting the same pick-location and orders are fulfilled faster. (Reid, 2024)

Cluster-picking is not to be confused with batch-picking, where the order-picker collects all the requested items in one batch at the same time. The batch-picking strategy requires a post-picking sorting step in which the collected items are assigned to their respective customer orders (Bartholdi and Hackman, 2019, p. 27), whereas in cluster-picking, items are collected separately for each customer order, which eliminates the need for further sorting and ensures efficient order consolidation.

1.1.2 Serpentine pick-path routing

The warehouse of the studied customer is arranged with parallel aisles and the routing strategy used is a *serpentine pick path* (also referred as an S-shaped path or traversal path) with potential for modification, see *Figure 1.1*. In a serpentine pick path, shelves and item storage locations are static and order-pickers are only allowed to travel in a one-way direction in each aisle, according to Bartholdi and Hackman (2019, p. 161). This narrow aisle-split design allows for a high density of pick faces. It also aids the order-picker to achieve a high picking efficiency when picking is characterized by high-volumes of a large quantity of SKUs. By guaranteeing to pass all item storage locations, serpentine pick-paths also reduce backtracking, or unnecessary retracing of steps, which minimizes directional confusion. The serpentine pick path used by the IMI customer can be modified in a way that allows one-directional horizontal movement between aisle ends. In *Figure 1.1*, the dotted line represents a serpentine pick-path, while the solid line represents movement according to the modified serpentine pick-path. For example, to pick item *C*, it is not necessary to travel all the way through aisle 2. The lines with arrows represent the following path options for the order-picker at the lower end of aisle 5. The order-picker can either enter aisle 6 by picking item G and return, following the green solid line, or by traversing the entire aisle 5 and pick item G while passing. In short, an order-picker can enter an oncoming aisle by the following options:

- **Full traverse** – Traversing the current aisle, picking as necessary, and entering the next aisle.
- **Return** – Traveling along the current aisle only as far as required to collect all necessary items, then returning to the same end from which the order-picker entered.

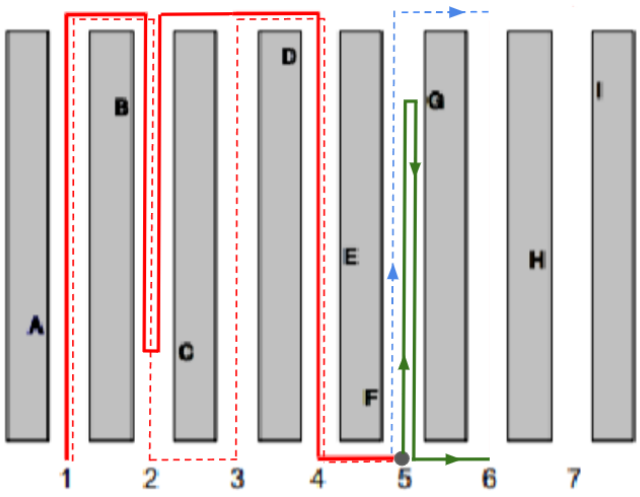

Figure 1.1: Representation of traditional (dotted line) and modified serpentine routing strategies (solid line).

1.1.3 Search algorithms

Search algorithms are methods used to systematically explore a problem space in order to find a solution. They are commonly used in fields such as path-finding, optimization, and decision-making systems (Korf, 2003). The main goal of a search algorithm is to locate a desired state or value among many possible alternatives. The efficiency of search algorithms is usually measured by the solution cost, the execution time, as well as the usage of memory.

In this thesis project, the algorithm execution time is limited to less than half an hour (interview, October 7, 2025). The main priority for the search algorithm is to find or generate the shortest possible picking route to reduce order-picking costs. Also, pick-orders are static, meaning that there is no arrival of customer orders over time, which is why the computational execution time of the model is allowed to vary. In the model, three different search algorithms were implemented to explore and find the shortest, or short enough, picking rounds by generating pre-configured unit-load configurations (ULC).

1.2 Industri-Matematik International AB

Industri-Matematik International (IMI) is a Swedish provider of warehouse management systems, and was founded by Martin Leimdörfer in 1967. IMI offers customized business solutions in all areas of warehouse operations, with a particular focus on inbound and outbound processes. IMI serves approximately 20 customers in fields that range from textile retail to third-party logistics (interview, October 7, 2025). In 2024, IMI reported a revenue of 237 MSEK (Allabolag). Initially, IMI operated as a consultancy firm, and in 1984, IMI released an in-house developed logistics system called ESS (Enterprise Strategic Solution). (Affärsvärlden, 1997)

The modules within the WMS of IMI offer tailored solutions depending on the different customer requirements and offer high flexibility to allow seamless integration with the governing ERP-system. One of the WMS-modules is called Pick Order Build+ (POB+), which automatically generates effective picking routes. POB+ uses algorithms that consider travel distance, routing-logic and customer-specific rules among other parameters to improve productivity and picking-efficiency. Customers who used the algorithmic module have reported improvements in picking efficiency of 3-10%. (Industri-Matematik International, 2025)

1.3 Current order-picking process

In the current cluster-picking policy provided by IMI to its customer, items ordered to a break-bulk terminal are put in a unique tote box at the warehouse, which is destined for an end customer in the downstream distribution network, see *Figure 1.2*. In the picking rounds, the boxes (also referred to as orders) can either be individually moved around by the order-picker, or more commonly, be loaded onto an empty unit-load which is moved around by a forklift in the picking lanes. In the thesis, a unit-load refers to a storage device used for transporting goods onto which boxes are placed. Common unit-loads include Euro-pallets, half Euro-pallets, and roll containers. As the order-picker successively completes and closes boxes, new boxes can be physically added to the unit-load. The unit-load typically needs multiple horizontal layers, as boxes vertically stack upon each other. The number of boxes per layer is limited by the dimensions of the boxes and the capacity of the unit-load surface. In addition, a layer constraint is imposed so that any box on a lower layer becomes inaccessible for picks once a box is placed directly on top of it. Re-stacking, meaning unloading and loading of boxes on the unit-load is not allowed.

After picking is completed for all boxes on a unit-load, it is consolidated and ready to be shipped out to the downstream customer. In the distribution network, the Break-bulk terminal receives the consolidated unit-load, which is subsequently split into its individual component tote boxes, which are then distributed to the end customers. The split could be done at a sorting terminal for a transport service provider or it could be done at certain drop-off locations at institutional buyers, such as hospitals or government agencies. (interview, October 7, 2025)

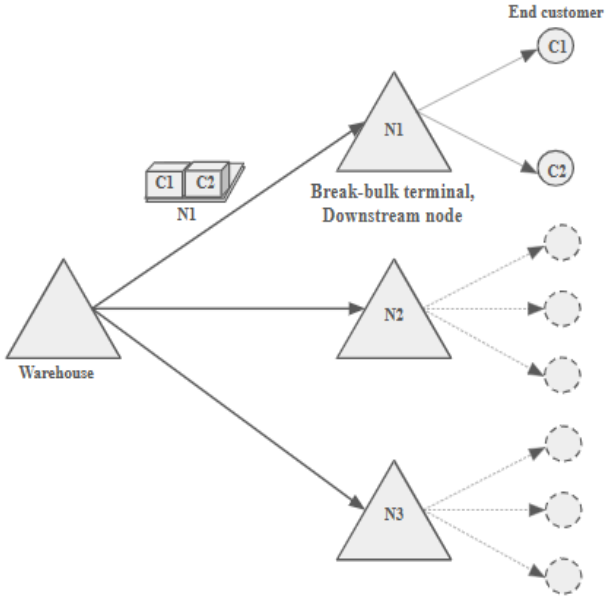

Figure 1.2: Illustration of the considered network. At the warehouse, orders are picked into tote boxes and put on a consolidated unit-load (e.g Euro-pallet). The unit-load is thereafter shipped to Break-bulk terminal N1, where the tote boxes are unstacked and sent to customers C1 and C2.

1.4 Problem description

The main problem is that the current picking procedure leads to inefficient box-stacking on the unit-loads. At present, the lack of planning on how boxes should be allocated on the unit-load often leads to the introduction of additional boxes, resulting in excessive time spent during picking and increased packing volume. A key factor is that a strict serpentine pick path is used, which entirely determines the stacking sequence of the boxes on the unit-loads.

The serpentine pick path can lead to inefficiencies when the order-picker has already moved passed one or more requested items from the pick-list, as not all boxes can be picked into simultaneously due to the layer constraint. Recalling that revisiting aisles that has already been passed in the current route is not allowed, the order-picker must then add a new box on the unit-load or re-enter the path in order to retrieve the item. This leads to excessive travel distance and travel time. As stated previously, it is acceptable for the pick path to be reentered multiple times to complete a pick-order, although, the number of picking rounds should be kept low to minimize travel time, total distance traversed, operator fatigue and operating costs.

Figure 1.3 illustrates a completed picking round that compares two distinct box arrangements in which travel has been made according to a strict serpentine pick path. Items that need to be picked in the same box are marked with the same number, and each layer on the unit-load is assumed to accommodate four boxes. For an inefficiently packed unit-load, the items have been

picked sequentially in boxes on the unit-load without any consideration being taken to forthcoming picks. As the order-picker traverses the path, boxes 1 to 4 are added to the first layer. Boxes 5 to 6 are subsequently added to the second layer, but the box with item 5 now limits access to future picks to box 1. To guarantee only one performed picking round, the order-picker must introduce a new box with an already existing order-number marking to the second layer in order to pick the remaining item 1. Finally, the last item can be picked to the accessible box with number 4. For the efficiently packed unit-load, the second layer is arranged so that boxes 1 and 4 remain accessible in the first layer for their next item picks. This is done by letting boxes 5 and 6 be positioned on top of boxes 2 and 3. This unit-load configuration (ULC) reduces manual handling and pick time by avoiding unnecessary box additions such as stacking an additional box 1 to the second layer.

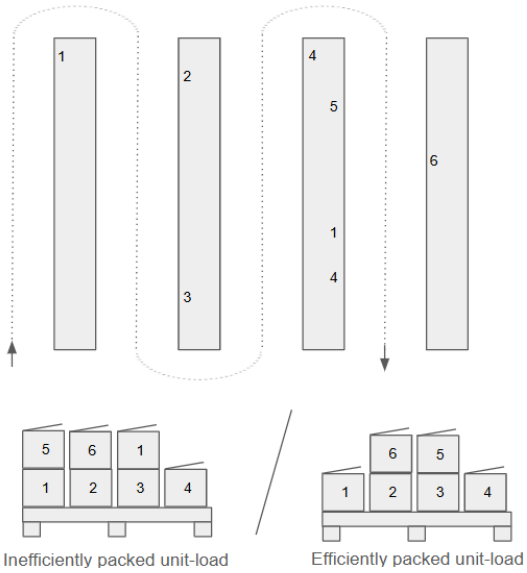

Figure 1.3: Example of a picking round with two different unit-load configurations.

An alternative solution to not introducing an additional box 1 in the example is to complete the picking round after finishing all the necessary picks for the boxes in the first layer. The order-picker would then reenter the path from the beginning and pick the items for boxes 5 and 6. However, this comes at the cost of traveling an excessive distance, as another full picking round must be performed.

The presented example assumed that only eight items had to be picked and that there are four boxes per unit-load layer. However, in reality there could be more than 200 ordered items in a pick-order and up to six boxes in seven layers to pack on a unit-load (interview, October 7, 2025). Clearly, variations in these parameters lead to a significant increase in the complexity of the sequence-constrained box-stacking problem. The related challenge is that the algorithm should be flexible and capable of generating unit-load configurations when these parameters vary.

To minimize the number of picking rounds and additions of unnecessary boxes, it is desirable to internally arrange the boxes on the unit-load in a way that allows the accessible boxes to be completed as early as possible. This is important because only a limited number of boxes (being equal to the number of boxes that can stack per unit-load layer) are accessible at a time and usually, a box must be turned inaccessible (i.e. closed and covered) before picking can begin for a new one. The challenge is to find the optimal way to internally arrange the boxes on the unit-load so that as many accessible boxes as possible are completed before adding boxes on the next layer. The goal is thereby to reduce the total time in the picking phase by minimizing manual handling such as introducing additional boxes with already existing order-number markings on the unit-load.

Beyond the challenge of planning box allocations on the unit-load, a related problem is that the current picking procedure relies solely on the strict serpentine path, which does not account for the available horizontal shortcuts between aisle ends. However, as discussed in Section 1.1.2, the order picker is allowed to exit an aisle after retrieving only the required items instead of traversing the entire aisle, which creates the possibility for substantially shorter picking routes. The associated challenge is to determine which routing decisions minimize the total travel distance for a picking round, while simultaneously respecting the precedence constraints imposed by the box stacking sequence.

The main challenges in the current picking procedure can be summarized as follows:

- **Unnecessary duplicate tote boxes are introduced -** Due to no future stack planning, boxes risk being made inaccessible for picks early in the route before all their items have been picked, forcing the order-picker to add new boxes of the same order-number marking.
- **Extra picking rounds are required -** When boxes become inaccessible, the order-picker must reenter the pick path and complete another full round, increasing the travel distance and picking time.
- **Lack of route planning -** The current routing strategy relies on a strict serpentine path, which prevents the use of shorter alternative routes and may lead to unnecessarily long travel distances and increased picking times.

1.5 Purpose

The purpose of this thesis is to develop a pick planning algorithm for the case company that reduces the total time in the picking phase by optimizing the picking route and box allocations on the unit-load.

1.6 Limitations

All the steps from analyzing the source code relating to the current picking processes to preparing for continued application of the algorithm will be covered in the thesis. However, the thesis will not include the implementation of the algorithm into the commercial version of the WMS. Also, the algorithm will be based on a list of assumptions that do not fully reflect all box-stacking possibilities on the unit-loads in practice.

2. Literature Review

This section presents the literature search procedure, an overview of existing research and theoretical framework that is relevant to the topics of this thesis.

The literature search followed the iterative process proposed by Höst, Regnell, & Runeson (2006). Initially, search of relevant literature began broadly using multiple keywords that related to precedence constraints, pallet-stacking, order-picking and search algorithms. After an initial screening of the abstracts of the identified literature, the search was iterated using more precise keywords and terminology to narrow down studies that were directly relevant to the thesis. Examples of such keywords were cluster-picking, S-shaped traversal strategy, box-stacking and precedence constraints. The narrow search of literature was also assisted by using references found in studies to attain additional sources of relevant information. The search portal LUBSearch was used, as well as scientific publishers such as Springer/Kluwer, Elsevier, and Wiley who provided access to several relevant publications.

2.1 Warehouse layout and routing strategies

In the comprehensive literature review "Design and control of warehouse order picking: A literature review" by De Koster et al. (2007) , the topics of layout design, order-picking processes and routing methods are covered. The objective of order-picking is defined as maximizing service levels while minimizing operational costs that arise from labor, machines, and capital. Order-picking can be decomposed into five subactivities; travel, search of items, picking, setup (such as constructing the pick list) and other. Of these activities, travel consumes the most of the time for order-picking, followed by search (consuming 50 % and 20% of the time respectively), see *Figure 2.1*.

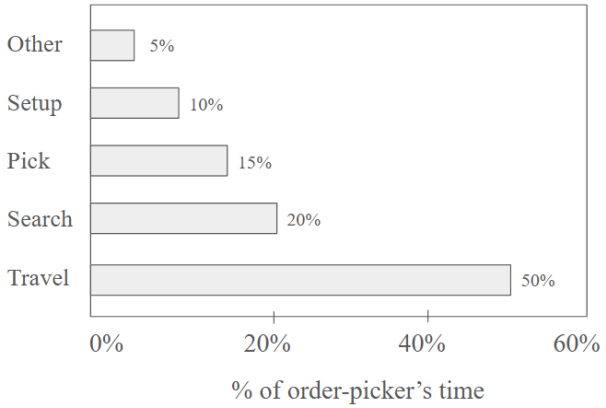

Figure 2.1: Time spent in each order-picking activity.

De Koster at al. also describe the goal of routing methods, which is to arrange the items on the pick list to minimize the traversal through the warehouse. The Traveling Salesman Problem (TSP) can often be applied for this type of problem. In an order-picking context, the TSP seeks to find the shortest possible route with the criterion that each location in a set is visited only once before returning to the starting point. However, the authors also mention that there is no guarantee that optimal routes exist, since different warehouse layouts dictate different operational constraints. Also, optimal picking routes might seem illogical to the order-picker, which leads the order-picker to deviate from the computed route. Various experiments have been conducted to compare routing strategies, such as the S-shape traversal strategy as well as the combined traversal strategy Petersen (1997). The results found out that the best heuristic solution was 5% longer than the optimal route on average.

To eliminate the need for finding the optimal route, the simplest routing heuristic is the S-shape or serpentine traversal strategy. Another simple routing strategy is the Return strategy, in which, an order-picker only enters aisles where there are picks, traversing them only as far as necessary, and returns to the aisle end after the pick. Roodbergen (2001) describes a combined strategy of an S-shape traversal strategy and return strategy, calling it a combined routing strategy. He states that strategies with simpler routing rules decrease pick errors and the time that the order-picker spends searching for the items.

Aboelfotoh, Singh and Suer (2019) aim to minimize the total distance traveled in a warehouse by efficiently assigning customer orders to batches. In the article case, pick-orders are static and an S-shaped routing strategy is used together with an cluster-picking strategy. However, the article examines the order-batching problem (OBP), which deals with the optimization of which customer orders to batch for a picking round to achieve a minimal traversal distance. Also, in the context of the thesis, the batches (corresponding to pick-orders) cannot be modified due to the scope of the study and the algorithm will thus operate downstream of the OBP. The study also considers truck departure times, which are not considered in this thesis, as time-related constraints regarding when customer orders must be picked within a pick-order are outside the scope.

The article states that S-shaped routing strategies with disallowed backtracking are useful for warehouses with many order-pickers as it reduces the risk of congestion in the aisles. Although the algorithm in this thesis is applied for cases of single order-pickers traversing the path at a given time, it could potentially be extended in functionality to accommodate multiple order-pickers, thereby increasing order-picking throughput. The article also reinforces the fact that travel distance governs the order-picking time, which strengthens the importance of optimizing pick-path strategies.

2.2 Generating Unit-Load Configurations

Three different search algorithms have been used to generate complete ULCs by exploring effective ways to combine orders for each layer. Search algorithms are suitable for this because they systematically search the solution space of possible box arrangements with the aim of finding an efficient or the optimal configuration.

2.2.1 Brute-force search

Brute-force (BF), or exhaustive search is an algorithm that evaluates all possible candidates to satisfy the predefined problem before concluding on the candidate with the best solution. Brute-force search is the most commonly used algorithm as it does not require any domain knowledge and will always find the best solution if it exists (Korf, 2003). All the algorithm needs is a goal state or value to find as well as an initial state. Brute-force search is best applied for problems of limited size where implementation simplicity is more important than the processing time. However, in reality many problems see a growing number of possible candidate solutions as the size of the problem, or solution space, increases. The main disadvantage with brute-force search is thus that combinatorial explosion of possible candidates is likely which quickly limits the algorithm efficiency. (Engati, n.d.)

2.2.2 Random search

Random search (RS) algorithms are heuristic approaches that explores the solution space by evaluating a subset of the solution space containing randomly selected candidates. Pure random search algorithms do not use any analytical model to consider the results of the candidates evaluated previously and they can attain their randomness from a pseudo-random number generator (Zabinsky, 2011). Global random search algorithms search across the entire solution space, while local random search improves a current candidate by searching nearby candidates for small improvements. However, finding the optimal solution is not ensured through random search, and finding it is essentially based on luck.

Random search can be useful for black-box optimization problems where knowledge of the system and internal structure between candidates is limited (Zabinsky, 2011) . The algorithm speed can also be easily controlled by deciding the number of random iterations to perform. Random

search can quickly generate a diverse set of candidate solutions and is therefore useful in large solution spaces where exhaustive search is impractical. (Bergstra & Bengio, 2012)

2.2.3 Branch-and-bound

Branch-and-bound (B&B) is a heuristic algorithm that systematically explores candidates in the solution space by decomposing it into smaller sub-problems, referred to as *branches*. The solution space for branch-and-bound can be represented by treelike structure, where multiple choices expand the solution space by branching out to it, creating sub-trees. The algorithm then recursively eliminates branches until a complete solution is achieved. In the worst case, branch-and-bound needs to explore all the candidates to find the optimal solution, effectively behaving like a brute-force approach. (Hillier & Lieberman, 2001 and Huang et al., 2010)

Bounding and pruning are key concepts to branch-and-bound as they reduce the search space. Bounding means estimating the best possible solution that can be achieved for a branch, and pruning refers to trimming off branches of candidates that are worse than the current best solution. Branch-and-bound is commonly used for IP-problems (*Integer Programming*), where decision variables are limited to integers, and the main advantage of the algorithm is that it can effectively eliminate large parts of the solution space that would otherwise be too exhaustive to explore. (Hillier & Lieberman, 2001 and Huang et al., 2010)

Roodbergen and De Koster (2009) use a branch-and-bound algorithm as a benchmark for generating the shortest picking route for a warehouse with parallel cross aisles. More precisely, they determine the shortest picking route by solving the TSP with a branch-and-bound algorithm. For this thesis, solving the TSP is regarded irrelevant since backtracking is not permitted and the relatively few movement options that the modified serpentine path offers renders full route optimization unnecessary. The article concludes that while the optimal shortest picking route can be calculated using the branch-and-bound, it might not be the most suitable in practice as the computation time is unpredictable. However, in this thesis, the warehouse layout is simpler than that of the study, which also included cross-aisles. Therefore, the B&B algorithm may not be required for path optimization, although, it could still be worth investigating its potential for generating unit-load configurations (ULCs), where identifying an optimal configuration might benefit from the effective use of the branch-and-bound approach.

2.3 Box-stacking under precedence constraints

Although the literature on pick-path optimization and search algorithms is very wide, there seem to be few articles which focus on deriving box-stacking principles on the pick-device/unit-load based on a routing policy. A reason for this is probably that many real-life applications of box-stacking need to consider apparent practical constraints such that the bin packing problem (BPP) aims to solve. The bin packing problem mainly deals with optimizing the arrangement of boxes with different dimensions to achieve efficient space utilization on a pallet (Ding, Fragkos and De Koster, 2018). To solve the bin packing problem, a number of parameters need to be considered,

such as varying box dimensions, weight limitations and pallet stability.

In the literature search, there was little success in finding any articles that considered "box accessibility" as a precedence constraint. Box accessibility in this context refers to avoid picking items for a new order (i.e. box) on the unit-load if it means to prevent access for picking items for an unfinished order. The identified articles addressing the thesis problem share similar constraints, but also examine optimal box allocation on pallets based on, for example, the 3D bin packing problem. In addition, the identified constraints are based on practical packing perspectives such as using box weight, fragility and thereby deriving an optimal picking route accordingly.

Žulj et al. (2018) propose a picker-routing strategy that uses the precedence constraint that "heavy" items must be picked before "light" items. The objective of the article is to reduce travel time in the order-picking process for a warehouse that also retrieves items by using an S-shape (serpentine pick path) routing strategy. In the study, two different subtours were used to describe the precedence constraint. The subtours were not restricted to follow an S-shaped path in the algorithm solution. A heavy subtour defined an optimal route in the warehouse for picking all heavy items and a corresponding light subtour for optimal picker routing of the light items. With the constraint that the heavy subtour must be completed before beginning of the light subtour, the optimization is determined by finding a combination of a heavy and light subtour that results in a minimum total tour length.

The method proposed in this thesis is inspired by the approach of decomposing the optimization problem, as is done in the study by Žulj et al. (2018). Since at least one layer is necessary on the unit-load, a precedence constraint could be to complete the current layer l, before adding boxes to layer $l+1$. This constraint prevents excessive towering of boxes in any position by allowing a maximum of one unfinished layer at a time. In addition, the algorithm can split the overall box allocation problem by optimizing box allocations for one layer at a time. However, while developing the box allocation logic within the algorithm, it will be investigated whether deciding box allocations layer-by-layer is a more effective strategy than deciding box positions for one box at a time.

Ding et al. (2018), investigate the problem of determining optimal picker routes and stacking sequences where items have stacking precedence constraints. Compared to the thesis, boxes are picked directly to the pallet which means that there is no split-case picking. Also, the stacking precedence constraint is based on box fragility rather than box accessibility. The authors also present an algorithm which aims to identify the visit sequence such that the pick distance and stack height are minimized. In their model, the objective is primarily to minimize the travel distance while minimizing stacking height as a secondary goal. The model uses a number of assumptions that are relevant also for the thesis, which are mentioned in the methodology section. However, their study employs an algorithm that solves the Traveling Salesman Problem (TSP), which is beyond the scope of this thesis. Additionally, the algorithm addresses the 3D bin packing problem, which, as previously noted, is outside the scope of this thesis.

After solving the TSP for picker-routing in the study, a second algorithm was used to improve the

first algorithm and further decrease the picking distance. In the second algorithm, three improvement operators were used, all while ensuring that precedence constraints are not violated:

(i) swapping full layers,

(ii) swapping boxes between layers, and

(iii) moving boxes to another layer.

Some of the improvement operations could be used in a similar way in the thesis algorithm to further refine the allocations of boxes after an initial generation of a unit-load configuration. For example, it could be investigated whether swapping certain boxes between layers and moving boxes to another layer results in more accessible picks overall.

3. Methodology

In this chapter, the methodology used to develop the pick planning algorithm is described. This includes the overall research approach, data collection procedures, model formulation, and computational implementation.

3.1 Operations Research Framework

The thesis will be based on a modified Operations Research, which has been described as a six step model by Hillier and Lieberman (2001, p.7). The distinction is that the model will not be described by a mathematical expression, as prescribed in step 2. Step 6 is not included in the scope of the thesis due to time constraints.

1. Define the problem of interest and gather relevant data.
A clear description of the problem should act as a guide throughout the duration of the project. By having established a clear problem definition, the problem can be further decomposed into smaller subproblems. Further, the collection of relevant qualitative and quantitative data play a vital role in a problem solving project. This is especially true for this thesis, where code will be developed based on data collected from customers. The available data might also have to be cleaned and prepared to suit methods, functions and other fundamental programming building blocks. Consideration must also be given to the availability of key people within the organization, as access to certain desired data at any point of the project might be restricted to a limited number of employees. Qualitative data will be collected mainly through an interview.

2. Formulate a mathematical model to represent the problem.
After the problem is defined, the next step is to formulate a mathematical model or idealized representation of the problem that represents the core of the problem. Hillier and Lieberman (2001, p. 10) explain that an objective function, defined in terms of decision variables, parameters, and constraints, can be minimized to obtain an optimal solution. The objective function provides a quantitative measure of performance and acts as a benchmark to evaluate alternative solutions. However, in the thesis case, it has been decided that a heuristic framework will be best suited to the problem. A heuristic procedure is a pragmatically designed solution approach that seeks to find an acceptable or near-optimal solution without guaranteeing optimality. Instead, the heuristic

approach focuses on efficiency and reasonable performance and tends to be used when cost or time is scarce and the model is very large.

3. Development of a computer-based procedure to derive solutions to the problem from the model.
After a careful assessment of the validity of the model formulation, the next step is to develop a computer-based solution. The procedure will likely contain search algorithms that generate minimal picker route solutions and unit-load configurations with corresponding costs. Accordingly, this thesis applies a heuristic algorithm to efficiently produce acceptable or near-optimal solutions. To develop the heuristic procedure and solution framework, it is key to analyze the current problem definition that includes descriptions of the warehouse layout, unit-load characteristics and constraints. This is done to ensure that the generated solutions respect the conditions and therefore are reliable.

4. Test the model and refine it as needed.
Given the complexity of translating an operational problem into a logical formulation, it is inevitable that the model will contain bugs and flaws. For example, computational complexity grows as more basic operations are executed in a sequence, which might slow down the algorithm if not considered. The model can be improved through model validation, which is done by testing the model, identifying major bugs and debugging them. Once the model outputs valid results, the model can be prepared for its application.

5. Prepare for the continued application of the model as prescribed by management
Once the model is complete, it is ready to be implemented in a system. In this step, the model is integrated with other necessary information systems and management becomes involved by deciding how to apply the model.

6. Implement
Finally, the model is implemented in the commercial version, which is a critical phase as this is the point where the project benefits are reaped. Continuous monitoring and performance evaluation are important to ensure that the model functions intentionally over time.

3.2 Interview study

In addition to collecting quantitative data from the database about current picking processes, there was also a need to collect qualitative data that could be used in the early stages of developing the algorithm. Therefore, an interview was conducted with an application consultant with experience in the WMS developed by IMI. The interview was useful because it provided some background information and general understanding of the problem area, which was needed before the algorithm design could begin.

Höst, Regnell, & Runeson (2006) propose three main types of interviews. An *Open-ended interview* seeks answers to a broader area of interest where the interviewee is encouraged to converse freely around the topic. This allows for greater flexibility and hands qualitative insights. A *Semi-structured interview* combines both open and standardized questions, which allows individual perspectives while maintaining a neutral approach to avoid reframing of questions. Lastly, a *Structured interview* is essentially an oral questionnaire with fixed questions and answer options. The aim is to improve clarity and reduce non-response, but requires more time.

It was decided that a semi-structured interview would be suitable to gain both a general understanding of the picking operations (such as the number of aisles, number of orders and number of pick-load carriers per picking round), but also for allowing the interviewee to pinpoint areas that could need special consideration concerning the algorithm. See Appendix A.1 for the interview guide, containing the questions asked to the interviewee.

3.3 Model assumptions

Several assumptions are applied to define the applicability and limitations of the model. These assumptions ensure that the algorithm runs in a well-defined and controlled environment. The assumptions are as follows:

- All boxes are rectangular and have equal dimensions.
- One box per order is adequate to accommodate all the required items.
- Aisles are equally dimensioned and contain an equal amount of shelves, which are equally wide.
- Weight and load limits for boxes and ULCs will not be considered.
- All orders and their associated items must be picked in only one pick-order.
- Pick-orders are static, meaning that the set of orders remains unchanged throughout the picking process.
- Backtracking, meaning that the order-picker revisits a visited item location in an already toured aisle, is not considered.
- The order-picker moves the forklift and picks items at a constant rate.
- During stacking, at most one box may rise above a fully stacked ULC-layer. Any additional boxes must be placed within the current ULC-layer and shall not result in a second box extending above the layer.

3.4 Modeling the pick-path and routing policy

As a first required step in the algorithm design, the warehouse was modeled as an abstract representation of the real warehouse, using only the properties that are relevant for this thesis. To model the warehouse layout and the item storage locations, a matrix containing shelf-objects was constructed, see *Figure 3.1*. The class Shelf contain pick-order line related information, and allow multiple pick-order lines to be assigned to the same storage location. In the matrix, the rows represent the positions of the shelves and the columns denote the aisle sides, where one full aisle has a left and a right side. An item storage location is marked with an order number that requires a pick at that location, or with zero if not.

To model the routing policy, the procedure was to adopt the principles of the pick-path optimization procedure presented by Bartholdi and Hackman (2019, p. 163). Their pick-path optimization adheres to the same operational assumptions as for the thesis case, with each item storage location needing to be visited once and no backtracking being allowed. A graph was constructed to represent the routing network where nodes, or decision points, at the end of aisles define feasible path alternatives to oncoming neighboring nodes. Given that the modified serpentine routing does not allow backtracking, a node cannot be a neighbor with a node from a previous aisle. For example, node B_{top} has the lower node B_{low} and top node C_{top} as neighbors. Solid lines mark the movement possibilities between the nodes. Note that picking rounds always start and end in the lower end of the path.

Aisles are formed between the top and bottom nodes of the same lettering. Traversing along aisle *B* enables picking from all shelves on shelf sides *1R* and *2L*, while the path between *start* and node B_{top} only allows picking from one shelf side, being *1L*. Similarly, picking can only be done from shelf side *3R* when traveling along the nodes *D*. It can also be observed that all horizontal movements and movements of *Full traverse* consume a constant distance.

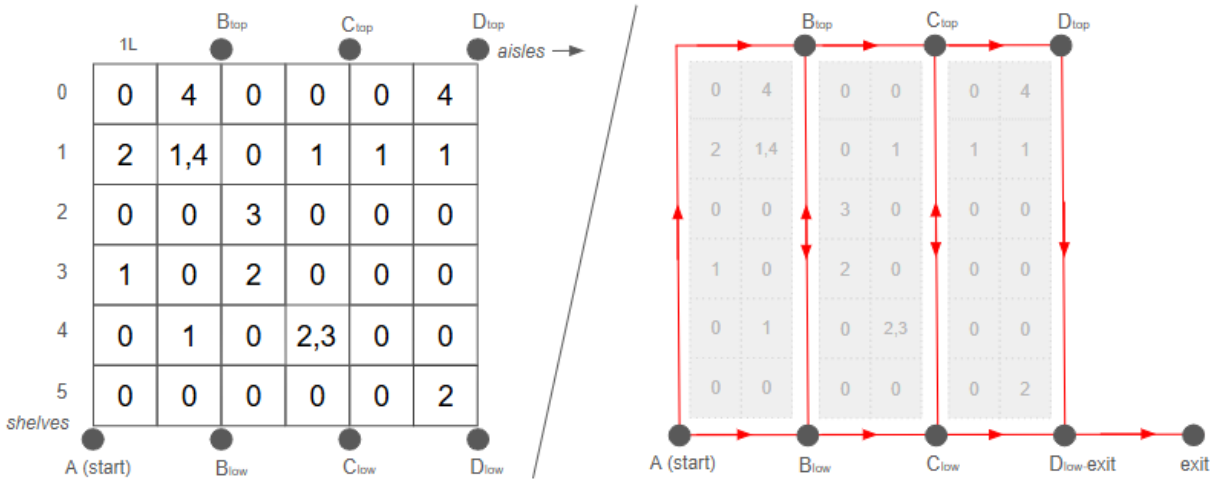

Figure 3.1: Representation of a fictive warehouse consisting of 2 full aisles with 6 shelves per aisle, a pick-order of four orders, and corresponding routing alternatives.

3.5 Input test data and preprocessing

Historical picking-related master-data is extracted from the customer database on demand, and structured in a spreadsheet-format. The available data contains information regarding the lay-out configuration of the customer warehouse, waypoints, item storage locations, picking-logs, pick-load carrier types, etc. Regarding the scope and assumptions in the thesis, it was irrelevant to consider data in pick-order lines such as order due dates, weight or volume. Also, the pick quantity and item pick height for each pick-order line could be ignored since the needed time for picking all items in a pick-order is assumed constant.

3.5.1 Warehouse mapping and collection of pick-order data

The aisle-specific warehouse input data that is relevant for the model was selected from the extracted master-data and can be seen in Table 3.1. The WMS represents warehouse elements and locations with a cartesian coordinate-system, where aisles, storage addresses and other warehouse elements are assigned an X-, Y-, and potentially a Z-coordinate for height measured in meters. *WSID* denotes the warehouse section that is subject to the picking operation. *Aisle direction* tells which axis the aisles are aligned in and *Direction pick* indicates whether the aisles are traversed from decreasing or increasing coordinates (illustrated by a minus or plus sign respectively) in the *Aisle direction* axis.

Regarding the *X-coordinates*, a range of values are used to represent the possible storage positions for items that are picked in a certain aisle. This is done to map the items to the corresponding shelf number in the warehouse layout matrix. However, since the X-coordinate can vary across storage positions that share the same shelf address, intervals are applied to slot the items to the correct shelf addresses in the warehouse model representation. Table 3.1 provides the mapping between X-coordinate intervals and their corresponding matrix column indices.

For example, aisle B starts at Y-coordinate 59.5 and ends at Y-coordinate 0, which is why it is picked in the Y-direction of decreasing Y-coordinates. For a processed pick-order, all items with X-coordinates that are in the range of $[1.25, 6.75]$ will be mapped to a corresponding shelf address in aisle B. In the warehouse matrix representation, column indices 1 and 2 correspond to the left and right side, respectively, of aisle B while row indices 0 to 120 indicate the mapping to the appropriate shelf address. For example, if the X-coordinate for an item lies within $[1.25, 4]$, then it is allocated to a shelf-position with the matrix column indice equal to 1, and if it lies within $[4, 6.75]$, it is allocated to a shelf-position with matrix column indice equal to 2.

Table 3.1: Pick-order line X-coordinates assigned to column indices in the warehouse layout matrix. Average X-coordinate values were calculated and used to mark the middle points of the aisles.

WSID	Aisle	Aisle direction	Direction pick	Aisle side	X-coordinate (range)	Y-coordinate (start-end)	Matrix column index
PP	A	Y	+	right	[0, 1.25]	[0, 59.5]	0
PP	B	Y	-	left	[1.25, 4]	[59.5, 0]	1
PP	B	Y	-	right	[4, 6.75]	[59.5, 0]	2
PP	C	Y	+	left	[6.75, 9.5]	[0, 59.5]	3
PP	C	Y	+	right	[9.5, 12.25]	[0, 59.5]	4
PP	D	Y	-	left	[12.25, 14.5]	[59.5, 0]	5
PP	D	Y	-	right	[14.5, 17.5]	[59.5, 0]	6
PP	E	Y	+	left	[17.5, 21]	[0, 59.5]	7
PP	E	Y	+	right	[21, 23.25]	[0, 59.5]	8
PP	F	Y	-	left	[23.25, 26]	[59.5, 0]	9
PP	F	Y	-	right	[26, 27.5]	[59.5, 0]	10

As for the Y-coordinates, the aisles A-G are all approximately 60 meters long. Based on the master-data for pick-order lines (see Table 3.2 for a selection of pick-order lines), the identified maximum number of unique shelf addresses, called *WPADR*, for aisles A-G was 249 and with a mean of 192 shelves per aisle. However, to maintain high mapping precision in the model while preserving computational performance, the number of shelves per aisle was set to 120. Consequently, each shelf was modeled with a length of 0.5 meters in the model, and it was decided that the aisle width should be uniformly set to 2 meters. In line with the aforementioned example, each item in the pick-order lines was assigned to the shelf address in the warehouse layout matrix that most closely corresponds to its X- and Y-coordinates.

In the master-data, there were eleven different warehouse sections (*WSID*), however, only *PP* displayed a layout that was translatable to the layout and routings in the thesis case. Other warehouse sections allowed two-directional picking, had aisles aligned in both X- and Y-axis or had varying aisle lengths. Note that aisle G, being traversed in increasing Y-coordinates, has not been included in the warehouse model because the serpentine routing always requires the path to finish in the lower end of the warehouse. Also, aisle H was excluded because it is shorter in the Y-direction than the other included aisles. Thus, the considered warehouse model consist of aisles A to F.

Data containing the coordinates for historical pick-order lines and their assigned pick-load carriers were also collected, see Table 3.2 for a selection of pick-order lines from the master-data including their system-designated names. *COID* represents the pick-order identity, originating from a break-bulk terminal and *PBCARID* represents the identity of pick-load carriers. *PBROWID* denotes the unique identifier of the pick-order line. The warehouse section (*WSID*) was filtered to *PP* to collect all pick-order lines from this part of the warehouse. This data was used to map the ordered items into the appropriate aisles and shelves in the described warehouse model matrix.

Table 3.2: Pick-order lines input example.

COID	PBCARID	PBROWID*	WSID	WPADR	X-coordinate	Y-coordinate
4666543	8789517	994	PP	A117A	0	34.4
4666543	8789525	009	PP	A081A	2.5	24
4666543	8789517	015	PP	A081B	2.5	24
4666543	8789517	974	PP	B118B	5.5	9.6
4666543	8789526	006	PP	C009A	11	3.2
4666543	8789520	013	PP	D054B	16.5	36.8
4666543	8789520	982	PP	F071A	24.5	26.4

Figure 3.2 shows the constructed warehouse with the required pick locations mapped approximately to their corresponding shelf locations. The mapping is based on the input data from Tables 3.1 and 3.2. Arrows indicate the movement directions within the aisles, while the row and column indices indicate how the X- and Y-coordinates of identified pick-order lines are translated into the warehouse layout matrix that is used in the model.

For example, the pick-order line with *PBROWID* being 015 in Table 3.2 has an X-coordinate being equal to 2.5. This places the item within the interval corresponding to matrix index 1, which represents the left side of aisle B (see Table 3.1). Its Y-coordinate then determines the corresponding shelf position of the item along aisle B. See *Figure 3.2* for its mapped location in the warehouse model.

*Only the last three digits of the actual *PBROWID* is shown.

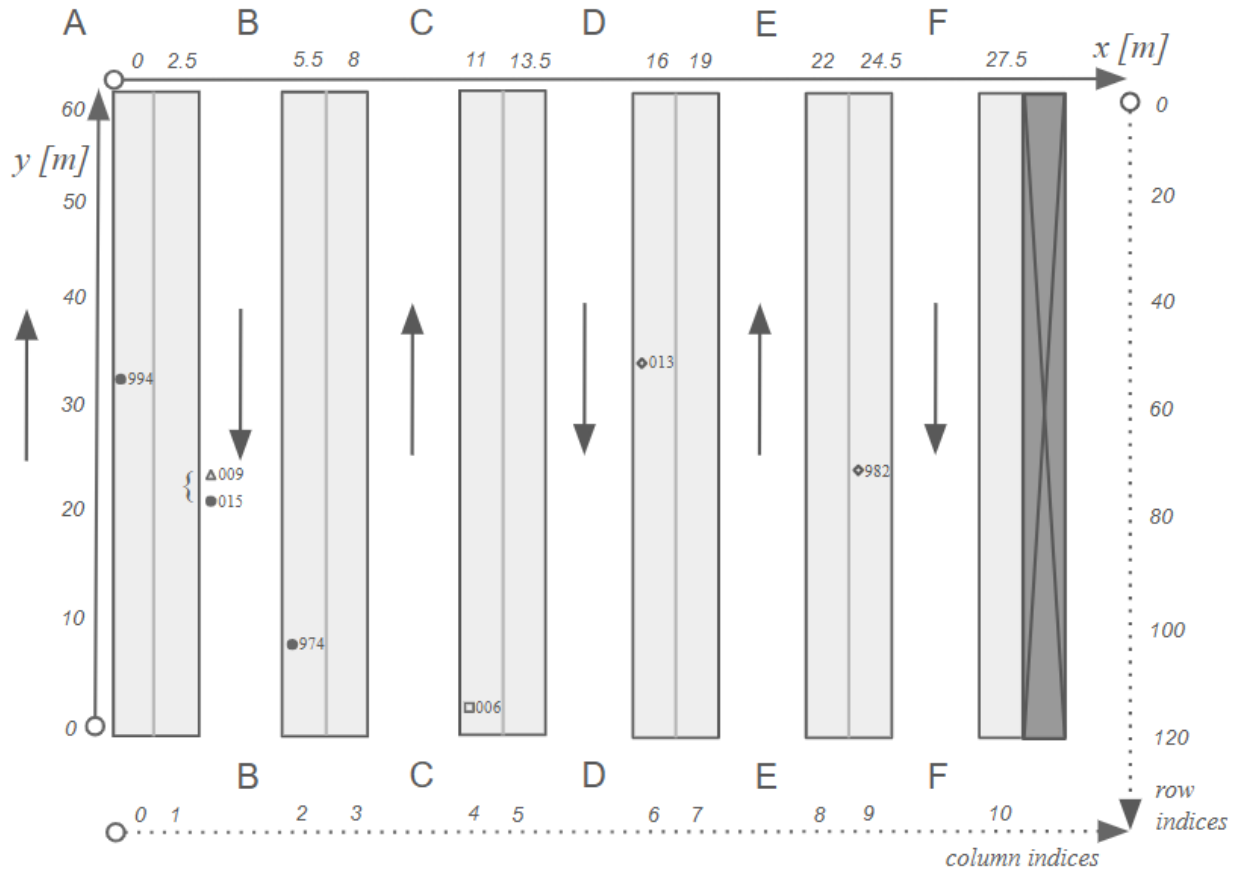

Figure 3.2: Representation of warehouse section *PP* in the model. Note that the pick-order lines presented in Table 3.2 have been mapped to their corresponding retrieval locations.

3.5.2 Extracted pick-orders

From the master-data containing pick-order lines, identified pick-orders were collected, see Table 4.2. These pick-orders met the operating conditions of the algorithm, meaning that the ordered items could be mapped into the constructed warehouse model in *Figure 3.2*. Regarding the collected pick-orders, a pick-order requested 14 pick-load carriers (PLCs) at most, and the number of pick-order lines (POLs) ranged from 20 to 29 in each pick-order. Furthermore, each pick-order had a minimum of 1 pick-order line per pick-load carrier and the maximum number of pick-order lines per pick-load carrier for was equal to 7. The average count of pick-order lines for a pick-load carrier (#POL/PLC) was equal to approximately 3 POLs when rounding up. See Table 4.2 for the case data that contain the identified pick-orders.

3.6 Unit-load box-stacking logic

To reduce the number of picking rounds, a viable approach is to perform an early screening of the pick-order to identify box-stacking possibilities. The screening can be done by, for each order/pick-load carrier, identifying the interval between its first and last occurrence in the serpentine path. Thereafter, box allocations can be planned by pairing together orders whose intervals do not overlap.

3.6.1 Order-sequence analysis

As the serpentine path fully determines the picking sequence, it can be interpreted into a linear order-sequence, as shown for a pick-order example in *Figure 3.3*. The first item for an order marks the start position, when the box is made accessible. Equivalently, the last item denotes the end position for the order, when the box is closed. The order-sequence shows the path positions for the ordered items in relation to each other. Items that need to be picked to the same box are marked with the same number. The order-sequence can then be further developed to a bar diagram (see *Figure 3.4*) that visualizes the span during which each of boxes 1 to 4 remains open (assuming that one box per customer order is sufficient). Let i denote the item position in the order sequence.

Assume that each layer on the unit-load accommodates two boxes. It can be seen that the *worst case scenario*, where the item pick locations are dispersed, will require two picking rounds to complete the pick-order (given the assumed box capacity for a layer), regardless of any pick path optimization. This means that swapping or moving boxes among unit-load positions has little effect on the total time during picking.

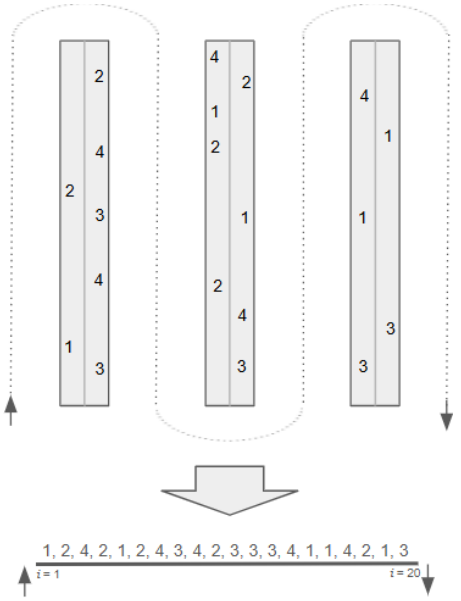

Figure 3.3: The serpentine pick path and its item storage locations can be interpreted into a linear pick sequence. Arrows indicate the start and end of the pick path.

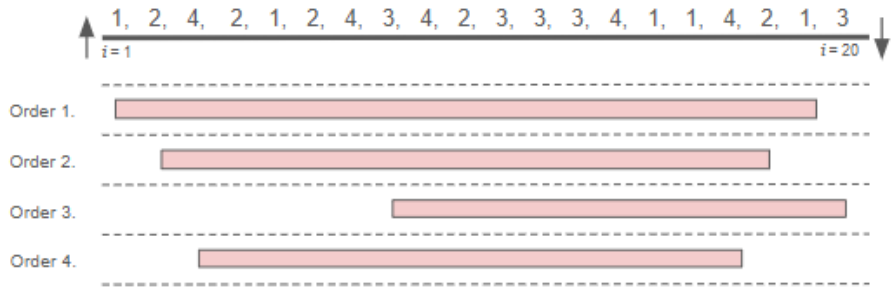

Figure 3.4: Bar diagram illustrating the spans where each box is open in the pick-order.

3.6.2 Order-stack

In *Figure 3.5*, an example of an order-sequence where two boxes can stack on top of each other is visualized. If order 1 is picked to the first unit-load layer, it will be completed before order 3 is encountered, which in turn can be stacked on top of order 1 in the same picking round. This relationship between order 1 and order 3 is called an order-stack (OS). Order-stacks can be useful for pick-orders with a high number of orders. This is because the number of required picking rounds is reduced by one if all orders in the first unit-load layer are part of an order-stack. *Figure 3.6* demonstrates scenarios for larger order-stacks (we call these $UOS_{3-orders}$ and $UOS_{4-orders}$), consisting of three and four orders. An order-stack consisting of four orders is essentially two concatenated order-stacks of two orders each.

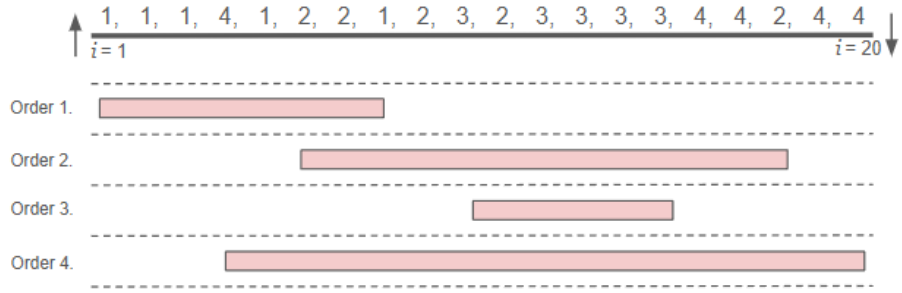

Figure 3.5: Bar diagram with an order-stack containing orders 1 and 3.

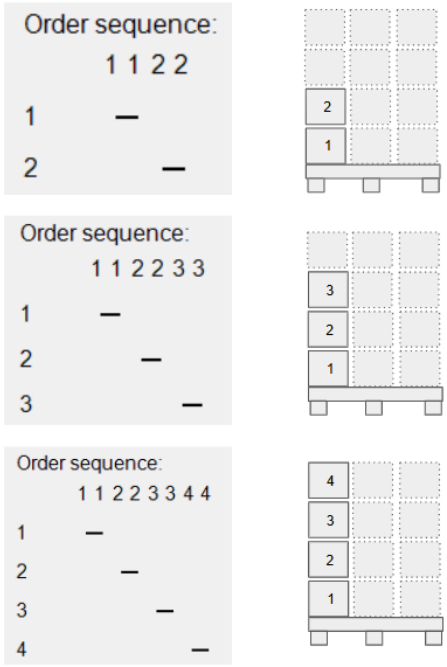

Figure 3.6: GUI representation with basic examples of standard UOS, $UOS_{3-orders}$ and $UOS_{4-orders}$. Includes corresponding examples of how they can stack on the unit-load.

In the model, each order had their start and end positions assigned in the serpentine pick-path by identifying their first and last item positions. For example, in *Figure 3.5* order *1* starts at $i = 1$ and ends at $i = 8$, and order *3* starts respectively ends at $i = 10$ and $i = 15$. If the first item position of an order is greater than the last item position of another order, then an order-stack is created. However, since there are many possible ways to construct an order-stack using the same order, this also comes with the risk that there is a possibility for multiple order-stacks to contain the same order, which is not allowed as an order can only be present once on the unit-load. Therefore, it is needed to create order-stacks so that each order is present only once in any stack. The decision rule is to keep the order-stack that contains the earliest occurrence of the orders in the order-sequence.

3.6.3 Unique order-stack

Another pick-order scenario where order-stacks contain the same order can be seen in *Figure 3.7* and Table 3.3. In this case, the correct pairing of orders into order-stacks is key to avoid multiple occurrences of the same order. It can be seen that there are three possible order-stacks in total, being OS *1-3*, *1-4* and *2-3*. However, by proceeding with OS *1-3*, it invalidates OS *2-3*, thereby allowing only one OS to be constructed in total. For an optimal solution, order-stacks *1-4* and *2-3* are chosen for box allocations, which will allow the pick-order to be completed in one round when traversing according to the serpentine path. Thus, by pairing orders into appropriate order-stacks, the number of order-stacks in the pick-order is maximized.

The procedure is to search among the identified order-stacks for an order that is only present once in one of them. Such an order-stack is called a unique order-stack (UOS). In the presented example, OS *2-3* is a UOS as order *2* is only part of this order-stack. By assigning box allocations with UOS *2-3*, the only remaining possible order-stack is OS *1-4*, which is also a UOS by the described definition. The pseudocode for the algorithm to create order-stacks and unique order stacks can be seen in Algorithm 1. Variables $o_{i,end}$ and $o_{j,start}$ refer to end and start positions for orders o_i and o_j in the order-sequence. $OS_{i,j}$ is an order-stack with o_i as bottom order and o_j as top order. Also, *orderStacks* and *uniqueOrderStacks* are sets that contain OS and UOS respectively.

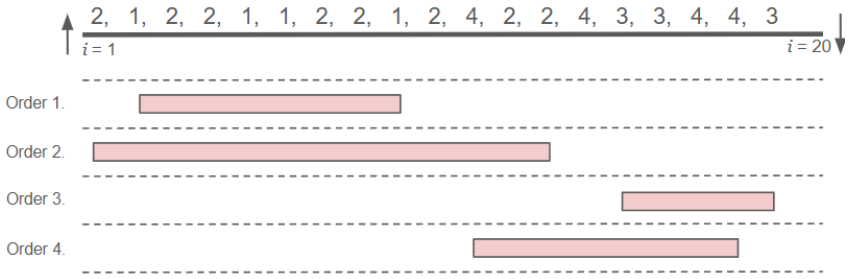

Figure 3.7: Order-sequence with order-stacks that include an identical order.

Table 3.3: Item positions for orders and possible order-stacks in *Figure 3.7*.

Order	i_{start}	i_{end}	OS participation
1	2	9	**3, 4**
2	1	13	**3**
3	15	19	**1,2**
4	11	18	**1**

Algorithm 1 Create unique order-stacks

```
function CREATEUNIQUEORDERSTACKS(O_all)
    for o_i ∈ O_all do
        for o_j ∈ O_all do
            if o_j,start > o_i,end then
                Create OS_i,j containing o_i and o_j
                Add OS_i,j to orderStacks
            end if
        end for
    end for

    sort orderStacks by o_i,start in ascending order    ▷ Prioritize earliest occurring OS
    for each OS_i,j ∈ orderStacks do
        if o_i ∉ O_used and o_j ∉ O_used then
            Add OS_i,j to uniqueOrderStacks
            Add o_i and o_j to O_used
        end if
    end for
return uniqueOrderStacks
end function
```

3.6.4 Limitations of the box-stacking logic

Although the box-stacking logic presented in Section 3.6 deterministically decides box allocations on the unit-load, it is oftentimes not possible to rely solely on this approach. This is because the box-stacking logic does not support partially filled unit-load layers, as an even number of orders is required for the unit-load to solely consist of unique order-stacks (UOS). If only the box-stacking logic is used for allocations, every order must be assigned to a unique order-stack (UOS). On the contrary, it is unlikely in most cases that there will be a sufficient amount of unique order-stacks present, why an additional box allocation method is needed to allocate orders/tote boxes which are not part of a UOS. For example, if a pick-order consists of 11 orders where there are 5 UOS in total, then 1 order must hence be allocated in a different way. The remaining order is therefore allocated using a pick-path algorithm in combination with search-based algorithms.

3.7 Pick-path algorithm

A pick-path algorithm has been modeled to manage pick-orders where the box-allocation logic is insufficient. The pick-path algorithm is required to find the shortest path to pick a set of orders in one picking round, considering the possible movement options at each node. The implemented algorithm is based on the algorithm described by Bartholdi & Hackman (2019, p. 163), which

also uses a node-link graph, as illustrated in *Figure 3.1*. The number of orders picked in a picking round is set to be at least equal to the number of orders (physically represented by tote boxes) per unit-load layer. However, more orders can be picked in a picking round if additional orders can be stacked on top of existing boxes (see Section 3.6). The algorithm is exhaustive, as it will evaluate every possible path before returning the shortest one. The exhaustive evaluation of the pick-path algorithm was believed to be computationally feasible given that each node offers at most two movement options leading to two other nodes. Had backtracking been allowed, the number of feasible paths would increase significantly due to the possibility to revisit already visited nodes.

The algorithm that finds the shortest path is a recursive depth-first-search procedure that evaluates all feasible neighbor edges from the current node in the graph. The pseudocode for finding the shortest pick-path algorithm is presented in Algorithm 2, and Table 3.4 describes the parameters used. In each iteration, the algorithm calculates the travel distance, d, which corresponds to moving from the current node, n_{curr}, to one of its neighboring nodes, $n_{neighbor}$. The distance d represents the minimum distance required to pick all items of the selected orders within the corresponding aisle segment, determined by choosing the lesser of the two movement options *Return* or *Full traverse* (see Section 1.1.2). For example, if there are many items spread evenly in the current aisle, it is likely that a *Full traverse* will be done. Conversely, if the order-picker only needs to pick one item that is located only a short distance into the aisle, then it is likely that the order-picker will return to the current node. A shortest path is found when the search reaches n_{target} (equal to the exit-node), at which point the globally minimal distance cost, d_{min}, is returned to the call stack. Initially, n_{curr} is set to the lower node A and n_{target} is set to the lower node F from the layout in *Figure 3.2*.

Table 3.4: Description of parameters in Algorithm 2.

Parameter	Description
n_{curr}	Current node
n_{target}	Target node (exit-node)
$n_{neighbor}$	Neighboring node to n_{curr}
$N_{visited}$	Visited set of nodes
d	Travel distance required to pick all requested items in the aisle between n_{curr} and $n_{neighbor}$
d_{tot}	Accumulated distance
d_{min}	Minimal distance for the picking rounds

Algorithm 2 Recursive Shortest Pick-Path

```
function FINDSHORTESTPATH(n_curr, N_visited, n_target, d_curr)
    if n_curr = n_target then                          ▷ Exit node has been reached
        return d_curr
    end if
    Add n_curr to N_visited
    d_min ← ∞
    for each n_neighbor do
        if n_neighbor is not in N_visited then
            Compute d from n_curr to n_neighbor    ▷ Movement according to Full traverse/Return
            d_tot ← FINDSHORTESTPATH(n_neighbor, N_visited, n_target, d_curr + d)
                                                                  ▷ Recurse from neighbor
            if d_tot < d_min then
                d_min ← d_tot
            end if
        end if
    end for
    return d_min                    ▷ Minimum distance found among all explored paths
end function
```

It shall be observed that the pick-path algorithm allows the order picker to return to the same node (i.e., the end of an aisle). This creates a potential risk of congestion with other order-pickers if the algorithm is implemented in a picking strategy involving multiple order-pickers simultaneously. This is an important risk to consider when the pick-path algorithm guides the order-picker, and can be avoided if the pick-order is completed using the box-allocation logic described in Section 3.6 (where the traditional serpentine traversal inherently prevents such conflicts).

3.8 Unit-load generation using search algorithms

Besides the graph representation of the warehouse, the described pick-path algorithm in Section 3.7 requires a set of orders as input to operate. This is because the pick-path algorithm calculates the minimal distance for the orders that are going to be stacked in one unit-load layer. For unit loads with partially filled layers, the number of orders to assign equals the remaining number of orders that are currently not assigned to previous layers. Consequently, the algorithm is executed once for each layer of the unit-load to pick all items in the pick-order.

For the pick-order, the challenge is to find sets of orders that result in minimal picking distances. The objective is to minimize the total distance, D_{tot}, to complete the pick-order. This total distance is equal to the sum of the distances, $d_{min,r}$, consumed in each picking round (see equation 3.1). R denotes the total number of required picking rounds.

$$Min(D_{tot}) = Min(\sum_{r=1}^{R} d_{min,r}) \tag{3.1}$$

3.8.1 Implementation of random search algorithm

The random search algorithm creates ULCs with randomly stacked box arrangements a given number of times (see pseudocode in Algorithm 3 and description of parameters in Table 3.5), and returns the ULC with the minimal found traversal distance. Firstly, all the required orders (that are stacked on the ULC) for the pick-order are rearranged in a random sequence. Afterwards, the sequence is divided consecutively into layers containing partitions of the orders. The distance corresponding to the picking round of each layer and its belonging orders is then calculated by using the pick-path algorithm and is then summed to the total distance of the ULC. Finally, the total distance of the ULC is compared to the current minimum distance, and if it is lower, it is recorded as the new minimum and thereby returned.

The number of random iterations of ULCs to generate, I, was set to 8000. Larger values of I rapidly increased computational time, and therefore this value was chosen to maintain acceptable computational performance. Confidence intervals will also be presented for the random search results to evaluate if the optimal solutions from brute force and branch-and-bound will fall within these.

Table 3.5: Description of parameters in Algorithm 3.

Parameter	Description
O_{all}	Set of all orders in a pick-order to be assigned
O_{seq}	Randomly shuffled sequence of O_{all}
k	Number of orders per layer
I	Number of random iterations to compute
D_{tot}	Total picking distance for a candidate configuration
D_{best}	Best total picking distance found so far
ULC_{cand}	Unit-load configuration candidate for lowest cost
ULC_{best}	Best unit-load configuration found (lowest cost)
d_{layer}	Picking distance for a single layer

Algorithm 3 Random search for best Unit-Load Configuration

function RANDOMSEARCH(O_{all}, k. I)
  $D_{best} \leftarrow \infty$
  for $i = 1$ **to** I **do**
    Create ULC_{cand} containing O_{all}
    Randomly shuffle O_{all} into a sequence O_{seq}
    Partition O_{seq} into layers of size k ▷ Form layers containing the partitioned orders
    $D_{tot} \leftarrow 0$
    for each layer $\in ULC_{cand}$ **do**
      $d_{layer} \leftarrow$ FINDSHORTESTPATH(n_{start}, $\emptyset$, n_{target}, 0)
      $D_{tot} \leftarrow D_{tot} + d_{layer}$
    end for
    if $D_{tot} < D_{best}$ **then**
      $D_{best} \leftarrow D_{tot}$
      $ULC_{best} \leftarrow ULC_{cand}$ ▷ Current ULC with minimal total distance
    end if
  end for
  return ULC_{best}, D_{best}
end function

3.8.2 Implementation of brute-force algorithm

The Brute-force algorithm (see pseudocode in Appendix A.2) calculates the total distance for every possible ULC and returns the ULC with the optimal distance. Firstly, the function *GetAllPartitions* creates all possible ULC-combinations. *GetAllPartitions* does this by initially generating all possible order combinations of size k, which is referred to as C (analogous to a layer of orders). A ULC-combination is built by recursively adding a new C and updating O_{rem} (set of remaining orders) until all orders have been assigned. Thereafter, the ULC is added to the set $candidates$, where each ULC_{cand} contains all required orders exactly once. Lastly, the pick-path algorithm is used and the ULC with the lowest minimal cost identified and returned. The algorithm steps are:

1. Generate all valid ULC_{cand} of O_{all} and of size k by calling *GetAllPartitions*.
2. In *GetAllPartitions*, a ULC_{cand} is created by recursively selecting a k-combination C from O_{all} and forming a layer.
3. For each complete ULC_{cand}, compute the total picking distance by using the pick-path algorithm to each layer and lastly sum the results.
4. Compare all generated ULC_{cand} and return the configuration with the minimum total distance.

3.8.3 Implementation of branch-and-bound algorithm

The branch-and-bound algorithm finds the optimal total picking distance and ULC by systematically exploring all the possible sets of orders while pruning branches that cannot improve the current best solution. The description of parameters is shown in Table 3.6. The steps are:

1. Compute the minimum picking distance for any single layer, d_{min}, by evaluating all k-combinations from O_{all}. A k-combination refers to a set of orders with the size k. This serves as a lower bound on the cost of any individual layer.

2. Recursively build a ULC one layer at a time. Before branching, compute $lb = D_{curr} + r \cdot d_{min}$. If $lb \geq D_{best}$, the branch is pruned.

3. For each candidate layer C, compute $D_{new} = D_{curr} + d_C$. If $D_{new} \geq D_{best}$, skip C. Otherwise, add C to the partial solution, recurse, and backtrack.

4. When all orders are assigned, update D_{best} if the complete solution improves upon the current best.

Table 3.6: Description of parameters in Algorithm 5.

Parameter	Description
D_{new}	Cumulative picking distance after adding layer C to a partially complete ULC
D_{curr}	Current total picking distance so far for a partially complete ULC
d_C	Distance associated with picking the orders in order combination C
L_{curr}	Set of layers built so far in the current branch of the search
O_{used}	Set of already assigned orders from O_{all}
lb	Lower bound on the minimal traversal distance of an ULC
r	Number of remaining layers

A more detailed explanation is provided hereafter. The pseudocode for the algorithm can be found in Appendix A.3. Initially, the layer containing the set of orders with the lowest possible picking distance, d_{min}, is calculated in the function *ComputeMinLayerCost*. The function iterates over every possible k-combination (C) from O_{all} and utilizes the pick-path algorithm to calculate the picking distance associated with picking the orders in C.

Thereafter, the recursive function *Search* is called, which builds a ULC one layer at a time. Each call updates the lowest possible picking distance and updates the number of remaining orders as well as remaining layers to build. Before branching, the lower bound, lb, is calculated as: $lb = D_{curr} + r \cdot d_{min}$. Since d_{min} is the minimum achievable cost for any single layer, lb provides an optimistic estimate of the best total cost that can be achieved from the current state. If

the condition $lb \geq D_{best}$ applies, then there is no existing solution in the sub-tree of the current branch that contains a lower cost than the current D_{best}, and the branch is therefore terminated early without further exploration.

If no pruning is performed, the distance for the next layer is calculated. In essence, the algorithm performs a depth-first search over all feasible layer combinations while using the bounding criteria to eliminate non-optimal partial solutions as early as possible, and so reducing the number of configurations that need to be evaluated. The distance d_C is computed for each order combination (C) of size k, and added to the distance corresponding to the current partial solution by $D_{new} \leftarrow D_{curr} + d_C$, see the pseudocode in Appendix A.3. Then, if $D_{new} \geq D_{best}$, the candidate layer C is skipped, as the total cost of the current partial solution is already equal to or exceeds the best complete solution found so far. The function *Search* is then called again recursively with the updated partial solution to explore all possible layer assignments that can be reached from the current state.

To allow the remaining sub-tree to be explored, C, and its containing orders are removed from L_{curr} and O_{used} respectively. This logic restores the state to its original condition before the next order combination candidate can be explored. Lastly, the algorithm returns the minimal distance when all orders have been used, and updates D_{best} to D_{curr} as new complete solutions are found.

3.9 Box allocation with unique order-stacks and search algorithms

The final step of the algorithm was to determine the framework for optimal allocation of orders to positions on the unit-load, based on combining the order sequence stacking logic and the search algorithms. It is hypothesized that allocations by placing identified unique order-stacks should be prioritized, as they enable more efficient stacking and inherently aim at reducing the number of picking rounds. Although, the allocation by unique order-stacks will likely only reduce the total picking distance more effectively when their allocations complete entire layers on the unit load, the steps are as follows:

1. If the number of unique-order stacks is sufficient to complete whole layers, which means $|uniqueOrderStacks| \geq k$, then assign *uniqueOrderStacks* to the unit-load.

2. If there are any remaining, unallocated orders, use search algorithms to generate efficient assignment of the orders to available positions on the unit-load.

3. If the number of unique-order stacks is insufficient to complete whole layers, $|uniqueOrderStacks| < k$, then use search algorithms to generate an efficient ULC containing all the orders.

The procedure tries to allocate unique-order stacks to the unit-load only if complete layers can be allocated with unique order-stacks (UOS). The reason is, as mentioned, that one traversal of the serpentine path then allows the order-picker to finish all the orders that are stacked on layers

where the orders are part of unique order-stacks. Optimally, it is desired that $|uniqueOrderStacks| \bmod k = 0$, which means, for example, that a pick-order with four layers could be completed in only two serpentine picking rounds, given that all orders are part of a UOS.

The explanation for why the larger unique order-stacks, such as $UOS_{3-orders}$ and $UOS_{4-orders}$ (see Section 3.6.2 and Figure 3.6), are not allocated is because it is hypothesized that the number of standard unique order-stacks decreases when a unique order-stack consisting of more than two orders (such as $UOS_{4-orders}$) are formed. This is since larger UOS might reserve orders that could otherwise be part of the unique order-stacks. It could therefore prove more beneficial to allocate the standard unique order-stacks as to prioritize filling layers and thus reducing the number of needed picking rounds. For example, if orders 1, 2, 3, and 4 can all be stacked together, then two separate unique order-stacks, namely UOS 1–2 and UOS 3–4, can be formed. Alternatively, a larger $UOS_{4-orders}$ (stacking the orders according to 1–2–3–4, where order 4 is on the top) can also be created. However, if $k = 2$ orders per layer, then two UOS would be more beneficial than the $UOS_{4-orders}$, as only one picking round would be required to pick all four orders.

Another reason why stacking with the large unique-order stacks containing, such as $UOS_{4-orders}$ would be inappropriate is that the constraint that maximum one box can tower over a fully stacked ULC-layer could be violated. This is the case if a ULC would consist of standard unique-order stacks as well as larger unique order-stacks containing $UOS_{4-orders}$. However, it could be of interest to investigate how allocation of larger unique order-stacks (such as $UOS_{3-orders}$ or $UOS_{4-orders}$) can benefit the algorithm.

Lastly, if there are no unique-order stacks for a pick-order, search algorithms are used to generate a ULC with the lowest possible picking distance. This means that the number of necessary picking rounds is equal to the number of orders per layer.

3.10 Development of GUI

A Graphical User Interface (GUI) was developed to increase usability and allow efficient interaction with the model. The GUI was developed in Windows Forms, and allows the user to input the required input data (see Table 3.7) to run the model. The user can input pick locations manually into shelves, use preset item locations or randomly generate item locations. It is also possible to select which of the search algorithms to apply for the inserted pick locations.

The purpose of the GUI is to provide a user-friendly environment by visualizing the linear order sequence, printing the generated ULC and visualizing the generated pick-path from a top-down perspective. The visualization of the pick-paths for a ULC is useful as it through a quick check allows for verifying that the required item pick locations are included in the generated route. However, the GUI will not be implemented in the final implementable version of the algorithm, as all interactions with the model will be executed within a module in the WMS called *PickOrderBuild*.

Table 3.7: Description of the necessary GUI input data to run the model through the GUI. The Data Type Shelf contain pick-order line related information.

Data	Data Category	Data Type
#Orders	Unit-Load Configuration parameters	int
#Orders per ULC-layer	Unit-Load Configuration parameters	int
#Aisles	Warehouse layout characteristics	int
#Shelves per aisle	Warehouse layout characteristics	int
Item storage locations	Inventory location data	Shelf[ ][ ]

3.10.1 Validation and test case design

As the algorithm was developed, continuous testing was done by setting up test cases to ensure that all selectable variables for the input were compatible and that the interaction in the GUI was correctly integrated with the graph and pick-path generation. Bar diagrams of the order-sequence was also made possible to allow the user to verify that unique order-stacks were correctly constructed. For the warehouse layout in *Figure 3.1*, the GUI representation of the shortest path found when picking for order 1 using the brute-force search algorithm can be seen in *Figure 3.8*.

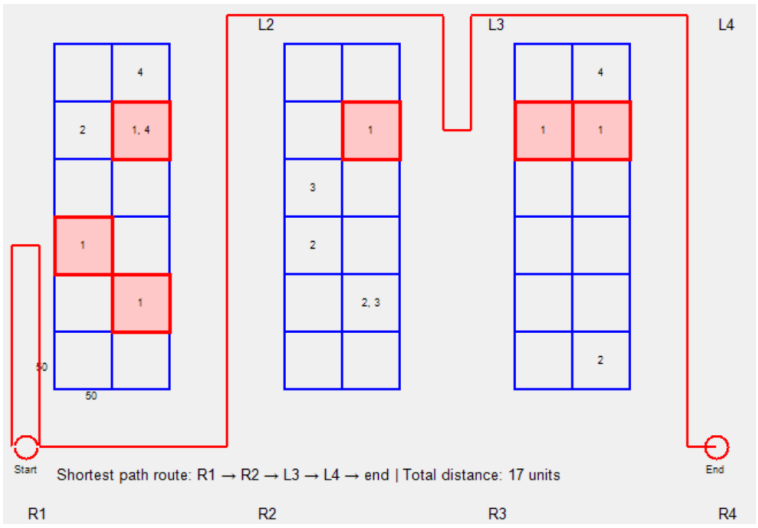

Figure 3.8: GUI representation of shortest pick-path traversal to pick all items in order 1 in a single picking round, as presented in *Figure 3.1*.

The dataset of the identified pick-orders, see Table 3.2, was regarded too scarce to fully test the efficiency of the algorithm. The main reason for this is that the maximum recorded number of orders/pick-load carriers (PLCs) was 14, while the maximum capacity of a unit-load configuration (ULC) can reach up to 42 PLCs in total where 7 layers each contain 6 PLCs (interview, October 7, 2025). It was therefore decided that additional pick-orders, or test cases, had to be generated to emulate larger pick-orders. To do so, further cases were constructed in which item

pick locations were generated in the warehouse according to a random distribution. The smallest number of pick-order lines (POLs) per PLC was set to 1, with additional generation ranging from 0 to 6 POLs, in each order to reach an average of around 3 pick-order lines per pick-load carrier. This ensured that each order was present at least once in a pick-order, and that the PLCs for the pick-orders shared similar characteristics as for the PLCs in the identified pick-orders. Data for the generated pick-orders are presented in Table 4.2.

3.11 Model output

The model output consists of instructions specifying which orders to pick in each picking round and the total number of picking rounds required, and which can be integrated to a pick-list. The model also outputs instructions as to which aisle end to travel next to. Further information normally included in a pick-list, such as item quantity, order due dates, and customer details will not be included in the model output. For pick-orders requiring more than one layer, the GUI displays the order numbers assigned to each layer. Where order-stacks are defined, the orders in (for example) layer 2 will be positioned on top of those in layer 1 in accordance with the box-stacking policy.

4. Results and Analysis

Analysis relating to the algorithm performance, as well as the captured results from the algorithm are presented in this part.

4.1 Evaluation of algorithms

This section evaluates the computational performance and reliability of brute-force (BF), branch-and-bound (B&B), and random search (RS) for varying number of orders, n, and orders per layer, k. In particular, BF is limited by fast combinatorial growth, making it infeasible for larger instances, while B&B reduces the search space by pruning poor solutions, although still facing scalability issues. RS remains computationally reliable, though at the cost of relying on pure stochastic search.

4.1.1 Computation time

The algorithms presented varying computation times and behaviors depending on the number of orders as well as the number of orders per layer in each pick-order (see *Figure 4.1*). The brute-force algorithm was able to run until the number of orders was equal to 13 orders and k was equal to 6 orders per layer, after which no results were obtained for higher values of n due to excessive computational time. The branch-and-bound algorithm started to drastically increase in time for pick-orders with more than 13 orders. Its computation time peaked at about 23 minutes for n and k being equal to 16 orders, respectively, 5 orders per layer. Generally, the branch-and-bound (B&B) algorithm was unable to produce solutions within a reasonable computation time for pick-orders exceeding 16 orders. Conversely, random search maintained a relatively constant computation time, with an average of little above 1 minute per run. This is probably explained by the fact that its computational complexity is equal to $O(I)$ where I denotes the number of iterations, which is easily controlled. Table 4.1 presents the computation times per case for all three algorithms averaged across 2,3,4,5 and 6 orders per layer. Dashes indicate cases for which no solution could be calculated within the time limit of 30 minutes.

Table 4.1: Average computation times per case, averaged across 2, 3, 4, 5, and 6 orders per layer. #cases denotes the total number of cases for which data was available. Dashes indicate no data available.

Case No.	#Orders	BF. avg	#cases	B&B. avg	#cases	RS. avg	#cases
1	8	0 s	5	1 s	5	52 s	5
2	10	1 s	5	10 s	5	1 min 16 s	5
3	11	9 s	5	15 s	5	1 min 19 s	5
4	12	6 s	3	3 min 7 s	5	1 min 9 s	5
5	13	40 s	1	3 min 3 s	3	1 min 31 s	5
6	14	-	0	12 min 0 s	2	1 min 34 s	5
7	15	-	0	17 min 5 s	2	59 s	5
8	16	-	0	20 min 8 s	1	1 min 24 s	5
9	20	-	0	-	0	1 min 20 s	5
10	30	-	0	-	0	1 min 49 s	5
11	42	-	0	-	0	4 min 15 s	5

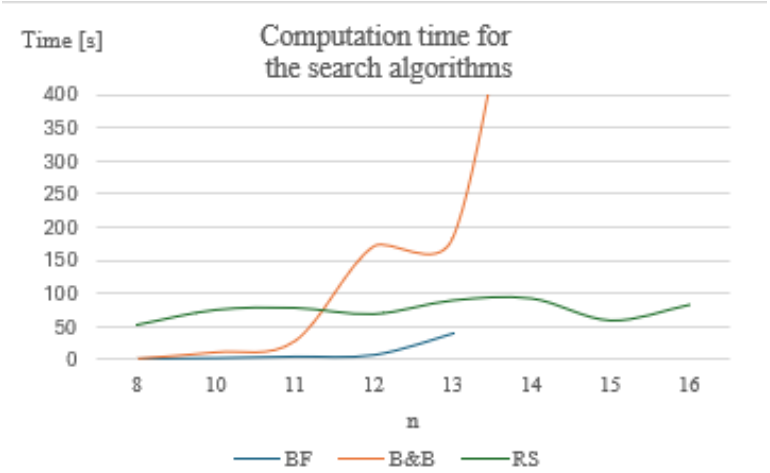

Figure 4.1: Average computation times of the search algorithms for cases with orders, n, ranging from 8 to 16, based on scenarios with two to six orders per layer. Derived from Table 4.1

Regarding the pick-path algorithm presented in Section 3.7, there are a number of factors that affect the computation time. Given that each node can have at most two neighboring nodes, there are never more than two possible paths from any given node, which limits how quickly the number of possible routes can grow. This property also gives the algorithm a complexity of $O(2^A)$, where A denotes the number of aisles. This is because at each aisle end, the algorithm effectively offers two possible routing choices, causing the total number of possible paths to grow exponentially with the number of aisles. In addition, the warehouse for which the algorithm is designed will consist of less than ten aisles, meaning that the computational time to generate potential shortest

paths will generally be relatively low. Other factors that affect the complexity are the number of shelves per aisle and the number of orders to pick to the unit-load.

4.1.2 Limitation of brute-force

As for the brute-force search algorithm, it is easy to realize that a brute-force search approach for the shortest pick-path would be too exhaustive for large pick-orders. This is because the number of orders to allocate to the unit-load as well as the number of orders per layer determine the combinatorial growth of possible unit-load configurations. The total amount of ULC-combinations can be calculated as in equation 4.1, which presents how to partition n orders into groups (layers) of size k.

$$\text{Total number of ULC-combinations} = \frac{n!}{(k!)^m \; m! \; r!} \tag{4.1}$$

The numerator $n!$ denotes all the possible permutations of n orders (imagine listing all orders in one long sequence and counting every possible ordering of that sequence). However, since the order of items within each layer is irrelevant, each group of size k (orders per layer) can be permuted in $k!$ ways so to avoid creating a new ULC-configuration. Therefore, $(k!)^m$ corrects for excessive counting across the m full layers. Additionally, the $m!$ term in the denominator accounts for the fact that the order of how the layers are arranged internally does not matter, effectively making sure that permutations with identical layers are not counted multiple times. Note that r = $n \mod k$ is the last layer holding the remaining group of orders. It can be observed that for ULCs without partial layers where $r = 0$, then $n \mod k = 0! = 1$.

For example, imagine a scenario of a ULC with $n = 12$ orders and $k = 4$ orders per layer. Using equation 4.1, we get:

$$\begin{aligned}\frac{n!}{(k!)^m \cdot m! \cdot r!} &= \frac{12!}{(4!)^3 \cdot 3! \cdot 0!} \\ &= \frac{12!}{4! \cdot 4! \cdot 4! \cdot 3!} \\ &= 5775\end{aligned}$$

It can be seen that there are 5775 possible ULC-combinations in the presented example. The number of possible ULC-combinations that the exhaustive approach would have to search through for some of the cases in Table A.6 (see Appendix A.4) can be seen in *Figure 4.2*, where n varies for a constant k, and conversely in *Figure 4.3* where k varies for a constant n.

In *Figure 4.3* It can be observed that the number of possible ULC-combinations that the BF needed to compute increased for lower values of k on average (see dotted line). This explains why BF did not calculate any results for any case with more than 11 orders when k = 2 orders per layer (see Table A.2), but succeeded with such for cases with 13 orders when k = 6 orders per layer (see Table A.6). The variation of k also likely explains the same limitation of B&B, although it being more effective by pruning poor solutions.

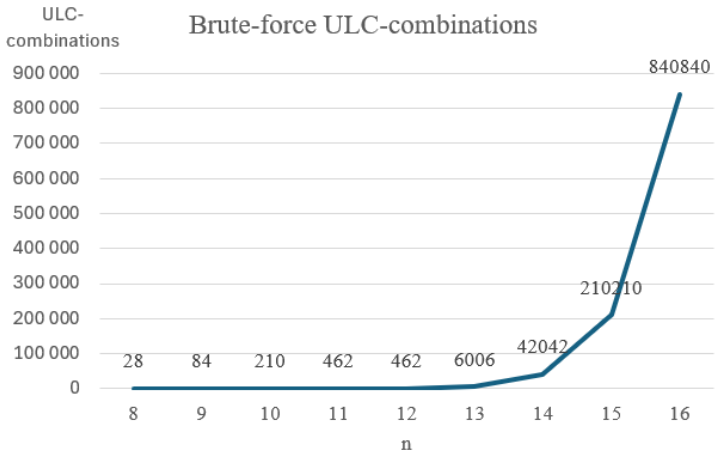

Figure 4.2: Demonstration of the number of possible ULC-combinations for different counts of orders, n, when $k = 6$ orders per unit-load layer. The number of possible combinations increase rapidly for increasing number of orders.

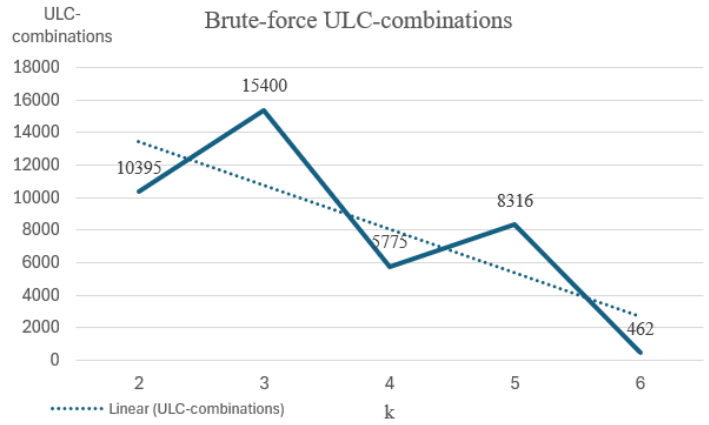

Figure 4.3: Demonstration of the number of possible ULC-combinations for different values of orders per layer, k, when $n = 12$ orders per unit-load layer. On average, the number of possible combinations decrease for an increasing number of orders per layer.

4.2 Computational results

For the capture of results, the algorithm has been tested for five different scenarios involving different numbers of orders per layer, k, ranging from two to six orders per layer. In each scenario, 11 cases with varying numbers of orders were considered. Cases 1 to 6 represent the identified orders from master-data while cases 7 to 11 represent the generated orders, involving larger numbers of orders. Overall, it should be mentioned that the algorithm allows any other non-zero value

of k. It can be mentioned that if $k = n$, then only a single picking round is sufficient to complete the pick-order as all the orders can be positioned on the first layer.

Using the warehouse layout derived from the input data, see *Figure 3.2*, the serpentine path distance required to complete one picking round can be calculated. Traversing all 6 aisles and 5 aisle widths results in a total distance of $(6 \cdot 60 + 5 \cdot 2)$ meters = 370 meters for a picking round, if not considering that the order-picker returns to the start point of the picking route (in which case, a picking round costs an extra $5 \cdot 2m = 10$ meters). The reason for not including this distance is that the order-picker can occasionally be required to continue picking to the current unit-load in another section of the warehouse. The serpentine cost for each case was then calculated as $Serp.cost = m \cdot 370$, where $m = \lceil \frac{n}{k} \rceil$ is equal to the number of unit-load layers and n refers to the number of orders in the pick-order. For example, a pick-order with 5 orders per layer and 12 orders in total will require $m = \lceil \frac{12}{5} \rceil = 3$ layers, which results in a total picking distance of $3 \cdot 370m$ = 1110 meters. The path cost corresponding to using the serpentine pick-path was chosen as the baseline, or *worst-case scenario*, for the comparisons between the cases.

The results describing the characteristics of the generated pick-orders can be seen in Table 4.2. The total traversed picking distances can be seen in tables A.2 to A.6, where the search algorithms and box allocation logic have been used on the cases where the orders per layer (k) vary. UOS+RS (unique-order stack + random search) denotes the box allocation procedure described in Section 3.9. It shall be noted that random search (RS) was used as the search algorithm to allocate the remaining orders, and that unique order-stacks were allocated even if their count were less than k. This means that unique order-stacks (UOS) were still allocated on the unit-load even when their number was insufficient to fill all k positions on a unit-load layer. As hypothesized in Section 3.9, it is only when the number of UOS is greater than k that a full picking round is saved. In the tables, dashes indicate that no solution could be calculated due to excessive computational time.

Distance costs and execution times are each reported as the mean of ten runs and standard deviations are shown as error bars for the runs using the random search algorithm. Execution times were measured on a workstation (Intel Core i5-14400, 32 GB RAM, Windows 11).

Table 4.2: Case data containing identified and generated pick-orders. Note that cases 7 to 11 represents the generated cases.

Case No.	#PLC	#POL	#UOS	Avg. #POL/PLC	Max #POL/PLC	Layers
1	8	20	3	2.33	4	2
2	10	26	2	2.28	4	2
3	11	26	5	2.27	7	2
4	12	28	4	2.36	6	2
5	13	28	6	2.15	4	3
6	14	29	6	2.07	4	3
7	15	77.3	1	5.16	9	3
8	16	72	2	4.5	4.50	3
9	20	81	3	4.05	7.7	4
10	30	87	6	2.9	6.0	5
11	42	96	6	2.28	4.7	7

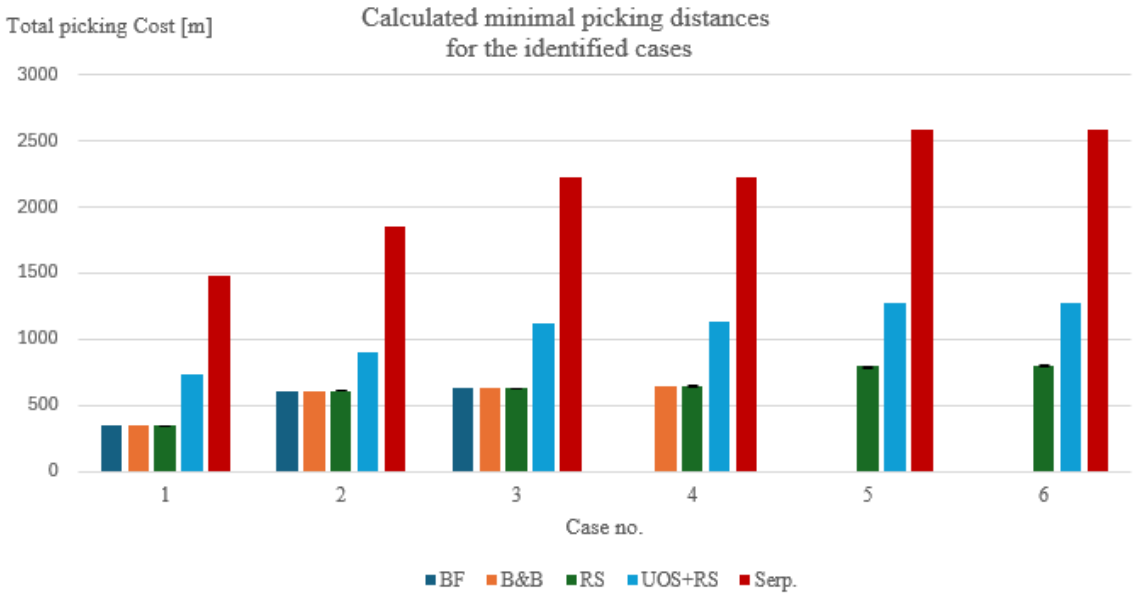

Figure 4.4: Calculated minimal picking distances for a ULC using the search algorithms for the identified pick-orders, with 2 orders per layer.

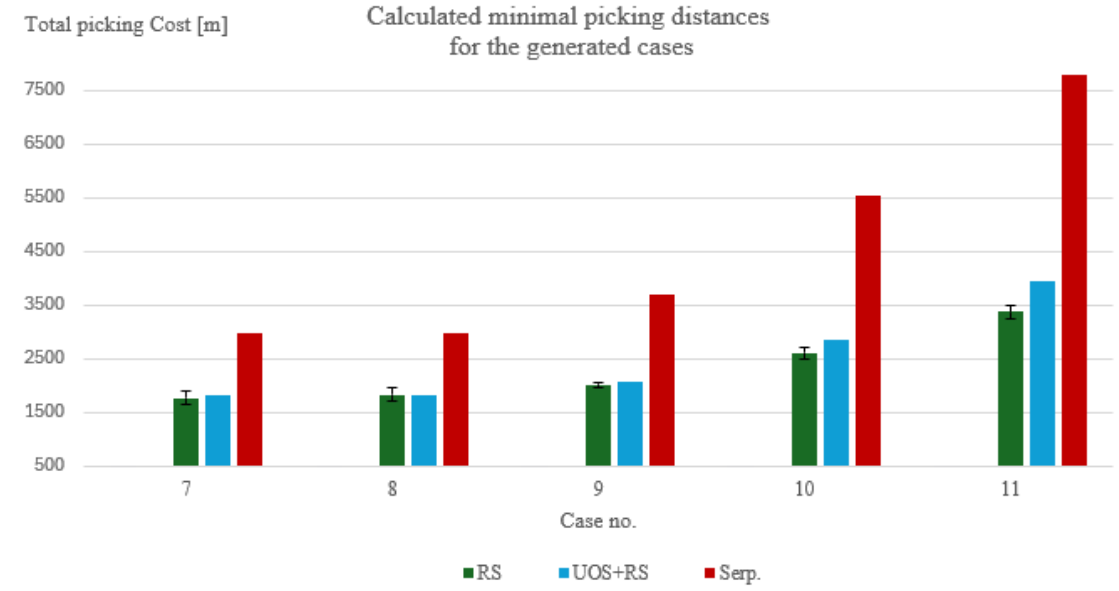

Figure 4.5: Calculated minimal picking distances for a ULC using the search algorithms for the generated pick-orders, with 2 orders per layer.

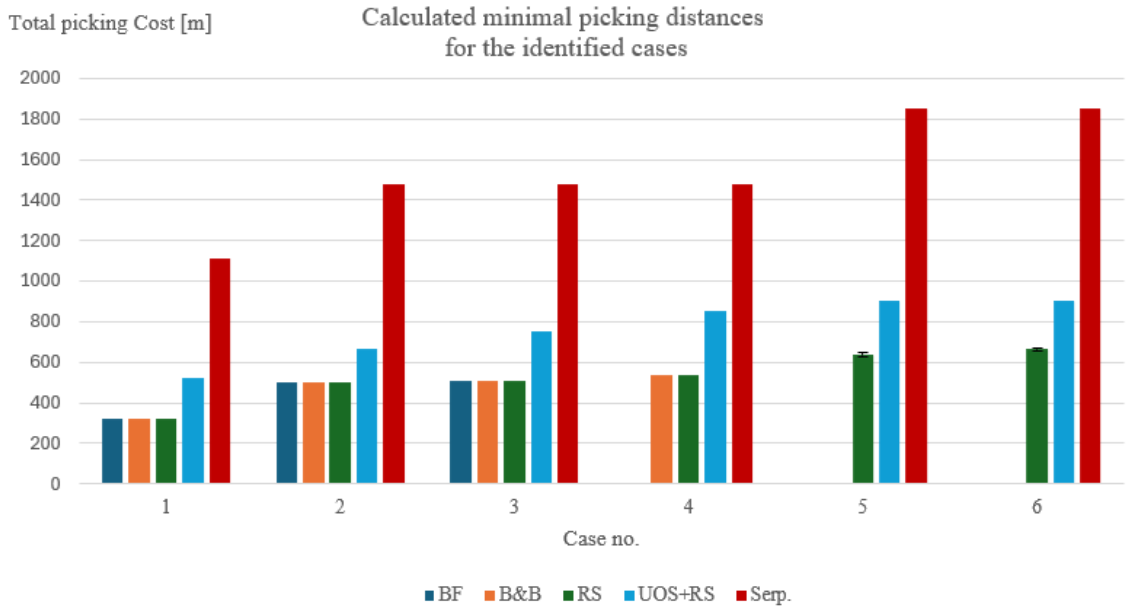

Figure 4.6: Calculated minimal picking distances for a ULC using the search algorithms for the identified pick-orders, with 3 orders per layer.

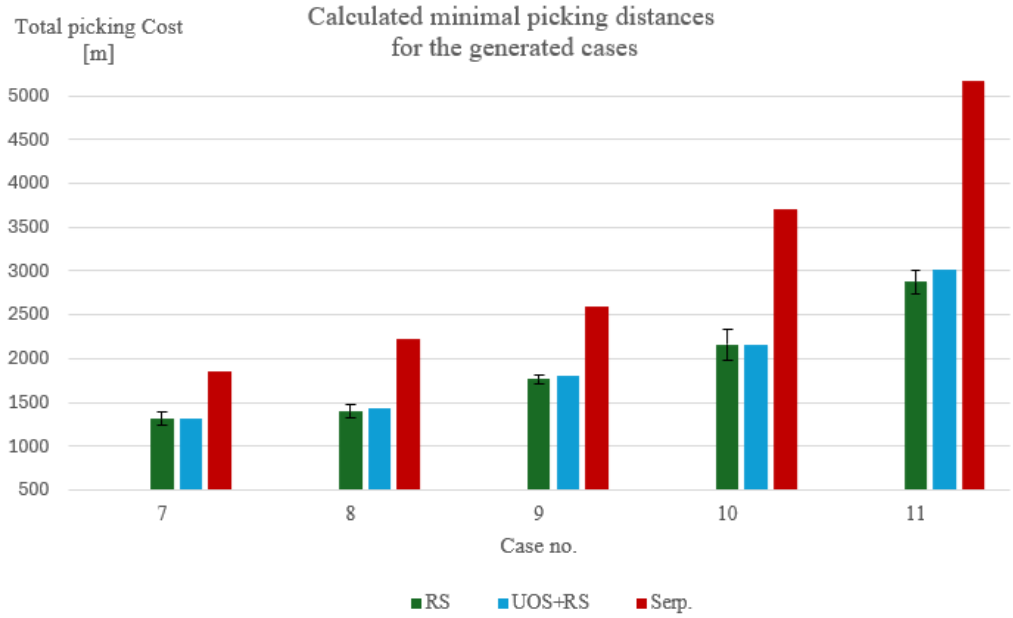

Figure 4.7: Calculated minimal picking distances for a ULC using the search algorithms for the generated pick-orders, with 3 orders per layer.

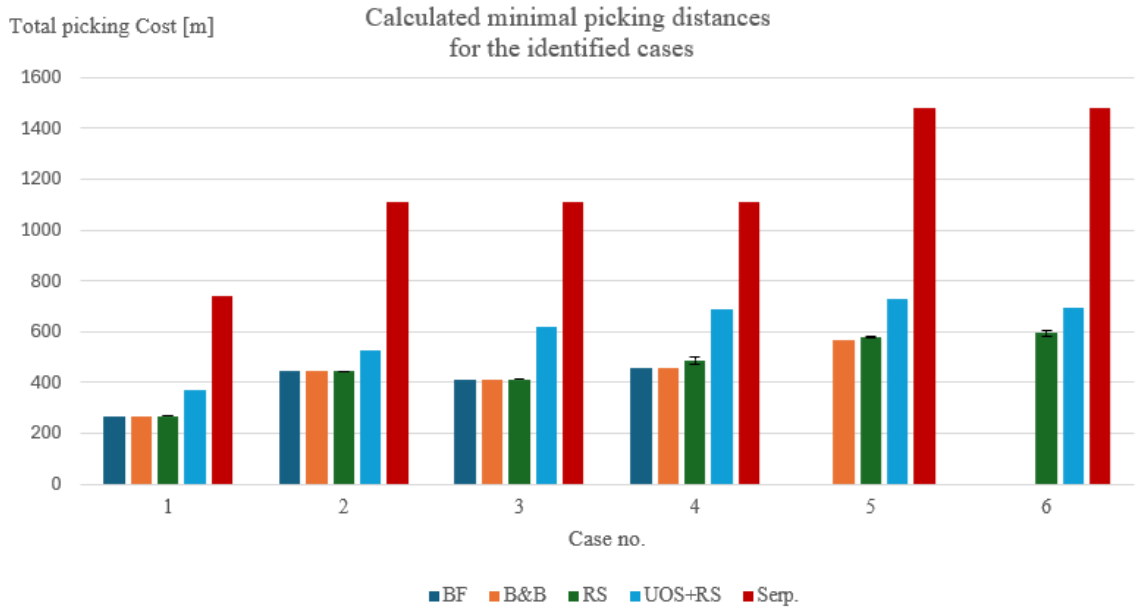

Figure 4.8: Calculated minimal picking distances for a ULC using the search algorithms for the identified pick-orders, with 4 orders per layer.

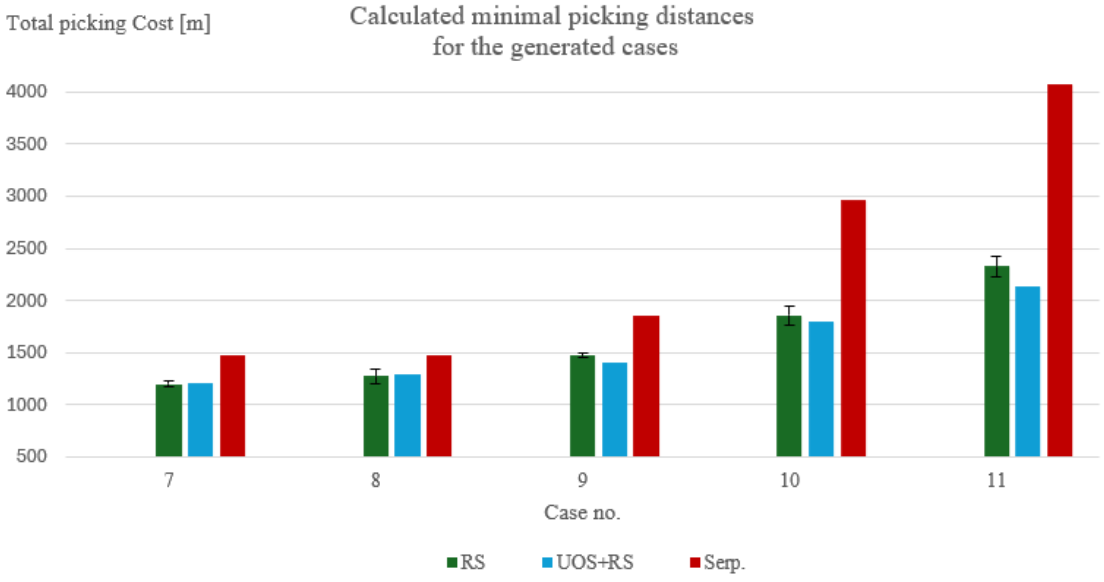

Figure 4.9: Calculated minimal picking distances for a ULC using the search algorithms for the generated pick-orders, with 4 orders per layer.

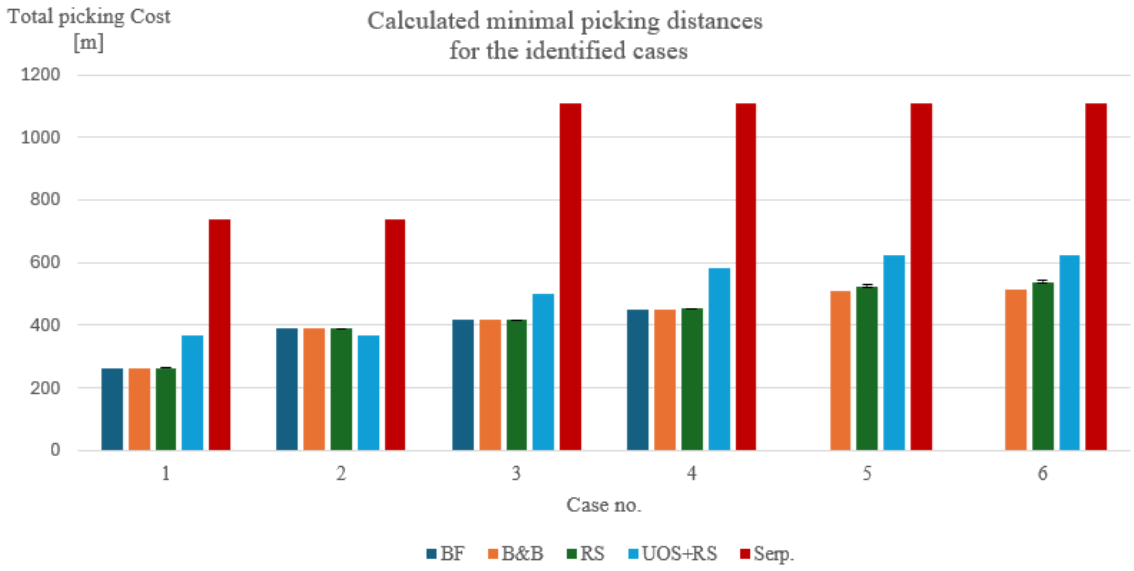

Figure 4.10: Calculated minimal picking distances for a ULC using the search algorithms for the identified pick-orders, with 5 orders per layer.

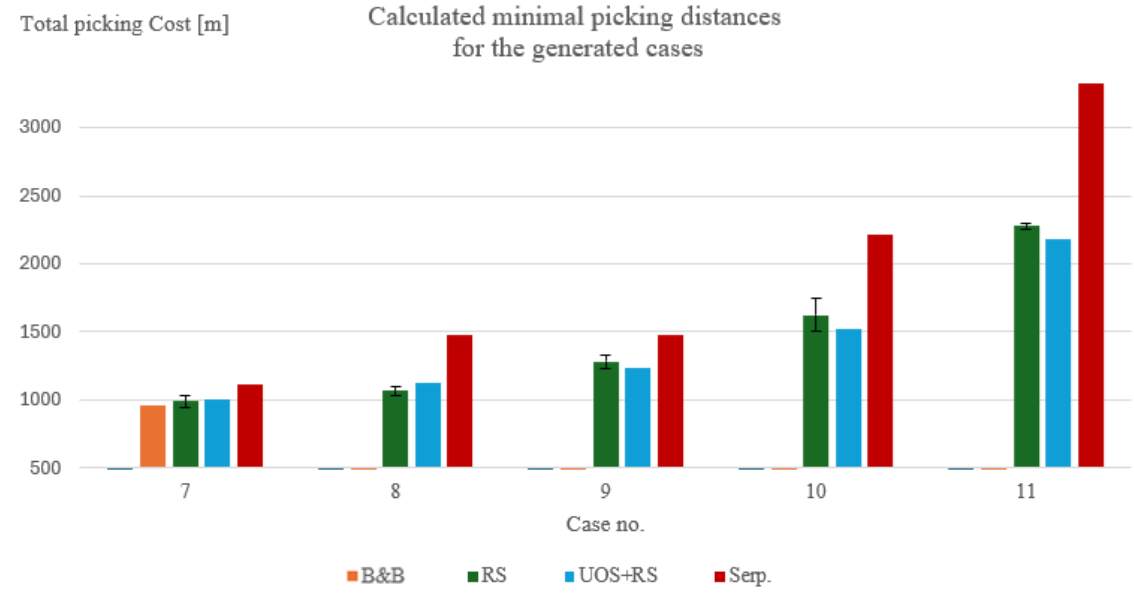

Figure 4.11: Calculated minimal picking distances for a ULC using the search algorithms for the generated pick-orders, with 5 orders per layer.

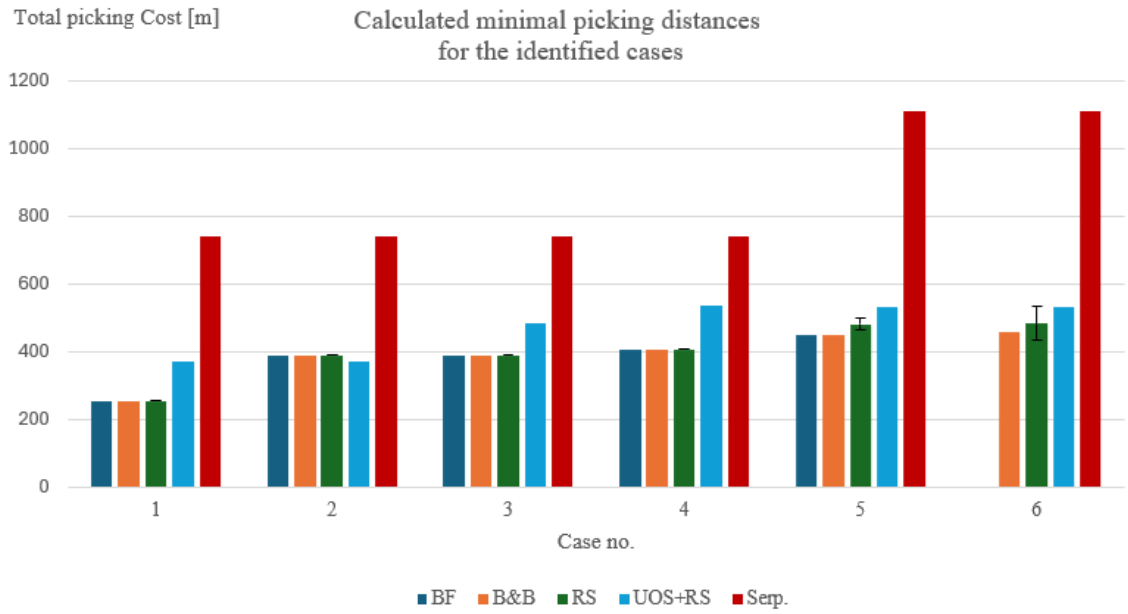

Figure 4.12: Calculated minimal picking distances for a ULC using the search algorithms for the identified pick-orders, with 6 orders per layer.

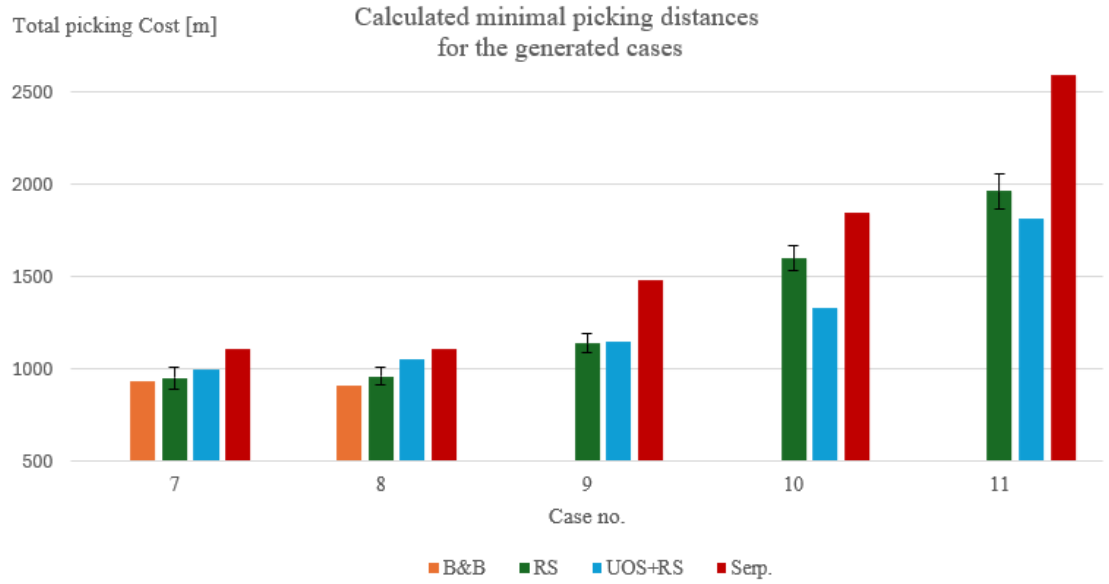

Figure 4.13: Calculated minimal picking distances for a ULC using the search algorithms for the generated pick-orders, with 6 orders per layer.

In Table 4.3, the calculated distances obtained from the search algorithms and the box allocation method (see *Figure 4.1* until *Figure 4.10*) were proportionally compared with the baseline (serpentine path distances) for each case, averaged over the different scenarios with varying orders per layer. In cases with identical improvement compared to the serpentine baseline path, the approach with the lower average computation time is highlighted.

Counterintuitively, RS generally outperformed the exact methods (brute-force and branch-and-bound) in cases 5 to 8. This is explained by the fact that not all scenarios provided a solution through BF and B&B when k varied. In these instances, only the scenarios that achieved a solution were presented. For example, B&B could calculate the best picking distance for case 8 in the scenario where k = 6 orders per layer, but not for other values of k in case 8 (see Appendix A.4). The baseline being 122% longer than the best found picking route is therefore derived from this single instance only (calculated as $\frac{1110}{910} = 122\%$), whereas for RS, it is calculated as an average across all available scenarios, where the improvement is generally greater compared to the baseline for lower values of k.

Table 4.3: Comparison showing how much longer (proportionally) the serpentine baseline path is on average than the calculated minimum picking distances across all cases and algorithms, for scenarios with two to six orders per layer. Note that the generated pick-order cases encompass cases 7 to 11.

Case	BF	B&B	RS	UOS+RS
1	**325**%	325%	325%	202%
2	**246**%	246%	246%	207%
3	**274**%	274%	274%	189%
4	223%	**258**%	254%	172%
5	248%	243%	**263**%	200%
6		230%	**258**%	202%
7		118%	**132**%	129%
8		122%	**138**%	134%
9			140%	**140**%
10			159%	**163**%
11			**173**%	171%

On average, RS outperformed the other methods in cases 5 to 8 and 11. The reason RS outperformed B&B and BF in cases 5 to 8 is likely due to that RS achieved relatively large improvements in picking distance in scenarios where $k = 2$ and $k = 3$ orders per layer across all cases. In contrast, B&B and BF were only able to run for a limited number of cases and scenarios with a higher value of k, meaning that their reported average results are based on fewer scenarios, being mainly those with larger k. In these scenarios, the improvements compared to the baseline tended to be smaller, which explains the lower average improvements reported for BF and B&B, and should not be interpreted as these algorithms calculating worse picking routes. In fact, wherever

BF and B&B did calculate a solution, the best picking routes were found.

4.3 Algorithm robustness

For the scenario where $k = 6$ orders per layer, 95% t-based confidence intervals and results were calculated in cases 6 to 11 (see Table 4.4) for the minimum traversal distances found by the search algorithms, see Table A.6. The reason is that the random search is stochastic and may produce different results over multiple runs. However, it can be observed that there is no variation in the solutions calculated by brute-force, branch-and-bound, and UOS+RS, as these methods are deterministic and therefore produce the same solution whenever the computation time limit is not exceeded. The confidence intervals serve to evaluate whether the optimal solutions found by brute-force and branch-and-bound fall within the range, providing an indication of how consistently random search finds the optimal solution over the runs. The confidence intervals were calculated according to $\bar{x} \pm (t^* \times SE)$ where $t^* = 2.262$ for when the degrees of freedom, df, was calculated to $df = n - 1 = 9$.

Table 4.4: Confidence intervals and results for the traversed distances calculated using the search algorithms for cases 6 to 11 where $k = 6$ orders per layer. Dashes indicate cases for which no solution could be calculated within the time limit of 30 minutes.

Case	RS 95% CI	BF	B&B	UOS+RS
6	[438; 522]	-	456	532
7	[802; 1068]	-	932	996
8	[836; 1054]	-	910	1050
9	[1021; 1243]	-	-	1147
10	[1354; 1648]	-	-	1328
11	[1710; 2130]	-	-	1816

Where BF and B&B were not possible to run because of excessive computation time, only RS and UOS+RS were able to generate ULCs. It can be deduced that RS is a more viable algorithm than UOS+RS for pick-orders where layers were not fully allocated with UOS for cases with less than 30 orders and where $k < 4$ orders per layer (see case 9 in Table 4.3). Although RS cannot guarantee finding the optimal solution, the 95% confidence interval in Table 4.4 indicates that the deviations from the optimal solutions remain small and show little variation across runs. The confidence interval show that the optimal solutions found by B&B fall within these intervals in all cases when $k = 6$ orders per layer. This suggests that RS can be a practically competitive alternative to the exact algorithms. Also, as the relative performance and variability between the search algorithms remained consistent across the other scenarios, the confidence intervals calculated for $k = 6$ orders per layer can be considered representative of the algorithm behavior across all scenarios.

5. Conclusions

Interpretation of results, key findings and comparison with literature is reflected in this chapter. In addition, the limitations and their effect on the algorithm are discussed here as well as directions for future research.

It can be ascertained that the model prevents allocation of multiple boxes for the same order on a unit-load, effectively resolving one of the main challenges. This is achieved because the pick-path algorithm systematically allocates orders on a layer-by-layer basis on the unit-load. As a result, boxes can no longer accidentally cover other boxes, improving manual handling efficiency and avoiding unnecessary traversals.

Given that all search algorithms in every pick-order case outperformed the serpentine pick-path (see Table 4.3), it can be established that allowing the *Return* movement significantly reduced the total picking distances for the pick-orders. The *Return* movement allows the order-picker to enter an aisle only as far as necessary to collect items before turning back instead of traversing the entire aisle. On the other hand, the strict serpentine path enforces the order-picker to traverse full aisles even if no item pick is performed in some of them. However, with the modified serpentine pick-path, called a combined routing strategy by Roodbergen (2001) and De Koster (2006), increased movement choices allow for shortcuts to be taken. It should also be noted that the improvements compared to the serpentine baseline route decreased as the number of orders to pick to the unit-load increased. For pick-orders with up to 14 orders with a varying k, the serpentine baseline was over 200% longer than the calculated picking route. For cases with 20, 30 and 42 orders, the serpentine baseline path was 140% (RS and UOS+RS), 163% (UOS+RS) and 173% (RS) longer than the calculated picking routes.

The method UOS+RS seems to be the better approach for order allocation when pick-orders contain more than 20 orders and where k vary between 4, 5 and 6 orders per layer (see Tables A.4, A.5 and A.6). For $k = 6$ orders per layer in cases 10 and 11, UOS+RS outperformed RS by 10% and 3% respectively (see Table A.6). In case 9, the total traversed distance was equal between RS and UOS+RS on average over all scenarios (see Table 4.3). For cases 7, 8 and 11, UOS+RS underperformed slightly compared to the search algorithms. However, in case 10, UOS+RS produced unit-load configurations (ULCs) with the minimum traversed distance, performing 4% better than RS. In this case, the reduction of one full picking round proved effective, even if it meant that the full serpentine path had to be traversed one time.

It is observable that the lower the number of allowed orders per layer is, the longer the calculated picking routes. This is likely because the algorithm enforces picking layer-by-layer, which means that the number of necessary picking rounds can be at most equal to $\lceil \frac{n}{k} \rceil$, where n denotes the number of orders and k the number of orders per layer. For example, if $n = 42$ orders and $k = 2$, there could potentially be 21 pick-rounds and layers on the unit-load, which is understandably excessive and unreasonable. It should be stated that in practice, k will most often be equal to 4, 5 or 6 orders per layer (interview, October 7, 2025). The reason why UOS+RS noticeably exceeded RS in this type of scenarios with lower value of k is because it could result in full serpentine traversals a number of times, depending on the number of UOS. In contrast, picking routes generated by RS allow shortcuts within each picking round, eliminating the need to traverse the full serpentine path and thereby reducing the overall picking distance.

For example, where $k = 2$ orders per layer and $n = 14$ orders, (representing case 6 in Table A.2) the picking route calculated by UOS+RS was 60% longer ($\frac{UOS+RS.cost}{RS.cost} = \frac{1272}{799} \approx 1.6$) than the one for RS (see Table A.2). In case 6, the number of UOS was equal to 6, which meant that there were $\frac{\#UOS}{k} = \frac{6}{2} = 3$ full serpentine picking rounds. This is because two layers were completed (allocated with tote boxes) in each round, as boxes could be stacked on top of one another when traversing according to the serpentine route. Thereafter, a remaining picking round was required to collect the items for the remaining 2 orders. This resulted in four picking rounds with a total picking distance of $3 \cdot 370m + 162m = 1272$ meters. RS proved that in this instance, it was more pick-path efficient to partition the orders into sets whose combined pick locations enabled shorter traversals within each picking round. See *Figure 5.1* for a visualization of the generated unit-load configuration for case 6 where $k = 2$ orders per layer.

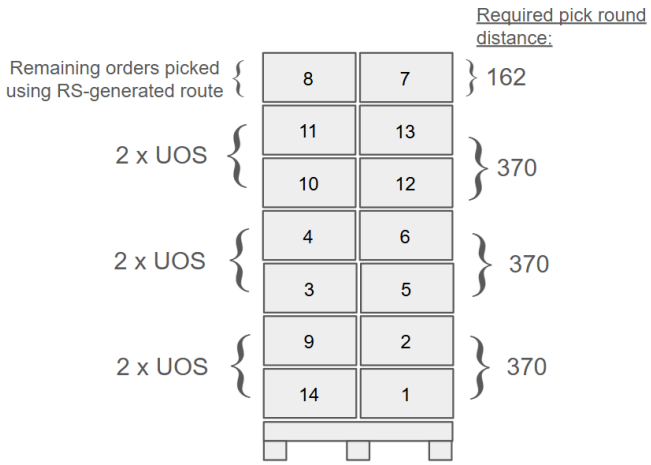

Figure 5.1: Visualization of a unit-load configuration generated using UOS+RS for case 6 where $k = 2$ orders per layer. Due to the six identified UOS, three complete serpentine picking rounds are required, with an additional final round needed to collect the remaining orders.

It should also be noted that the number of created UOS seems to be proportional with the number of orders in a pick-order. A possible explanation is that during the generation of items in the warehouse, the frequency of orders containing only a single item also increased. Presumably, single-item orders were common for pick-orders with many UOS. For the identified pick-order in case 5, (see Appendix A.5), 6 out of 13 orders were single-item orders, which meant that their box accessibility spans were minimal, and hence, facilitated for UOS-creation by pairing them with other orders. In conclusion, UOS+RS is the preferred method for box allocation for large pick-orders when the number of UOS is sufficient and k is either 4, 5 or 6 orders per layers.

The optimal picking route could be found by BF up until case 5 when the number of orders was 12 and $k = 6$ orders per layer, see Table 4.3. Although BF was marginally faster than B&B in most cases, B&B managed to find the optimal traversed distance in instances/cases with more orders (value of n) than BF could over all scenarios. The reason why BF was faster on average than B&B for pick-orders with fewer orders (see cases 1-5 in Table 4.1), is likely because the total number of ULC-combinations is also small, meaning that BF can enumerate all candidates very quickly while the branching and pruning of B&B adds unnecessary operations (such as computing a lower bound before each branch) without providing any noticeable reductions in the search space. However, as the number of orders grows, the branching and pruning process of B&B becomes increasingly valuable, as it managed to find the optimal traversed distance in instances with more orders than BF could handle across all scenarios. For example, in Table A.4, BF was able to compute the best picking routes within the given computation time only up to case 4, whereas B&B succeeded in finding the best routes up to case 7. Consequently, brute-force can be considered an inefficient algorithm for finding the optimal ULC, whereas branch-and-bound is more effective for larger orders due to its lower computational time, as it generally identifies the optimal solution within the time limit of 30 minutes while brute-force fails to do so. When k was equal to 2 or 3 orders per layer, B&B found the optimal route until case 4 (See figures 4.4 and 4.6), and when k was equal to 6 orders per layer, it found the optimal route until case 8 (see Figure 4.13). However, it should be noted that the computation time for B&B varied drastically in certain scenarios, making it rather unstable.

For randomly generated pick-orders, where the number of pick-order lines (POLs) per pick-order as well as the average number of pick-order lines per pick-load carrier (#POL/PLC) overall increased drastically compared to the cases of the identified pick-orders. This could be explained by the fact that because item locations were randomly distributed across all shelves and aisles, the items to be picked were more widely dispersed in the warehouse. This dispersion of items is likely the main reason why the cost of the search algorithms was relatively closer to that of the serpentine path in the generated cases compared to the identified cases. The higher number of POLs for the generated pick-orders could also be the reason for why less unique-order-stacks were present in cases 7 to 11 on average in relation to the number of orders. As more items for an order are distributed randomly in the warehouse, the probability of extending box accessibility spans increases (remembering that the box accessibility span is determined by the position of the first and last item pick in the serpentine path). All things considered, it can be concluded that the more widely distributed the items are in the warehouse, the longer the total traversal distance for a picking round will be and the lesser the improvement will be compared to the baseline.

This finding is also in agreement with the discussion of optimization worthiness by Bartholdi and Hackman (2019, p. 169).

Regarding computational complexity of the algorithms, it can be deduced that the pick-path algorithm has a limited contribution to the total time consumption since the number of aisles in the cases was equal to six. Of the search algorithms, BF has the highest computational complexity, having to calculate every candidate unit-load configuration (ULC) in the solution space. This makes it unpractical for pick-orders with more than 13 orders. B&B generally performed better than BF by pruning infeasible solutions, however being unstable in computation time and limited for pick-orders of 12 orders (if $k = 2$ orders per layer) and 16 orders (if $k = 6$ orders per layer). The time it took B&B to find the optimal picking route increased drastically after this number of orders. Lastly, RS had a more predictable computational complexity that could be practically adjusted by changing the number of random ULC-combinations to explore. The computational complexity for UOS+RS was in parity with that of RS for the same reason. Subsequently, it can be concluded that BF and B&B are not the most appropriate search algorithms for ULC-generation, and it is worth investigating if local improvements can be integrated to RS, or perhaps implementing another type of search algorithm.

5.1 Recommendations

Based on the fact that the model solves the practical issue of adding unnecessary boxes to the unit-load and enables the planning of every traversal, its implementation is recommended for IMI. However, it would need further adaption before doing so as the current assumptions and operational constraints must be considered at least to some extent in practice. At the very least, the algorithm and the basis for box allocations could serve as a platform for the development of a more tuned pick-planning algorithm.

In its current form, the model presents a framework in which the choice of box allocation is determined by the number of orders as well as the number of orders per layer. It is recommended that box allocations on the unit-load should be performed using the B&B algorithm for the following cases: pick-orders with up to 12 orders where k is equal to 2 or 3 orders per layer, pick-orders with up to 13 orders where $k = 4$ orders per layer, pick-orders with up to 15 orders where $k = 5$ orders per layer, and pick-orders with up to 16 orders where $k = 6$ orders per layer. As for UOS+RS, it should assign orders to the unit-load when there are 20 or more orders and k is equal to 4, 5 or 6 orders per layer. In the other instances, the minimally traversed distance should be calculated by RS. A decision tree for the recommended procedure can be seen in *Figure 5.2*.

To further increase order-picking efficiency for the IMI customer, it is recommended to reevaluate the composition of pick-load carriers assigned to pick-orders when more than one unit-load is necessary to complete a pick-order. This reconsideration would categorize as an order-batching problem as it would allow one to select which pick-load carriers to complete in each picking round. In such a scenario, a feasible strategy would be to reassign pick-load carriers across the pick-orders and in that way construct as many unique-order stacks as possible for the pick-orders.

Another strategy could be to segment the pick-load carriers by grouping carriers with item pick locations that are concentrated in the same warehouse area. Solving the order-batching problem with this objective can be effective in reducing the traversed distance for pick-orders with high item dispersion. However, it should be pointed out that the constraint that only a single ULC is sufficient to complete a pick-order must first be exempted from the algorithm, which then would need to be adapted appropriately. Alternatively to solving the order-batching problem (OBP), influencing a change of the storage assignment policy of the IMI customer could be a viable way to reduce potential item dispersion and thereby increasing the number of created UOS.

An advantage with traversing the serpentine pick-path is that directional confusion is reduced, and the order-picker can focus on remembering item pick locations instead of planning the fastest route to the next item pick. However, the combined routing strategy would necessitate the order-picker to possess an instruction as where to perform the next item pick, either being visualized in a map, or perhaps conveyed as pick-by-voice. To implement the algorithm, it is key that order-pickers receive item pick instructions before or during each picking round from the WMS. It is therefore recommended to use the algorithm output to provide instructions to the order-pickers.

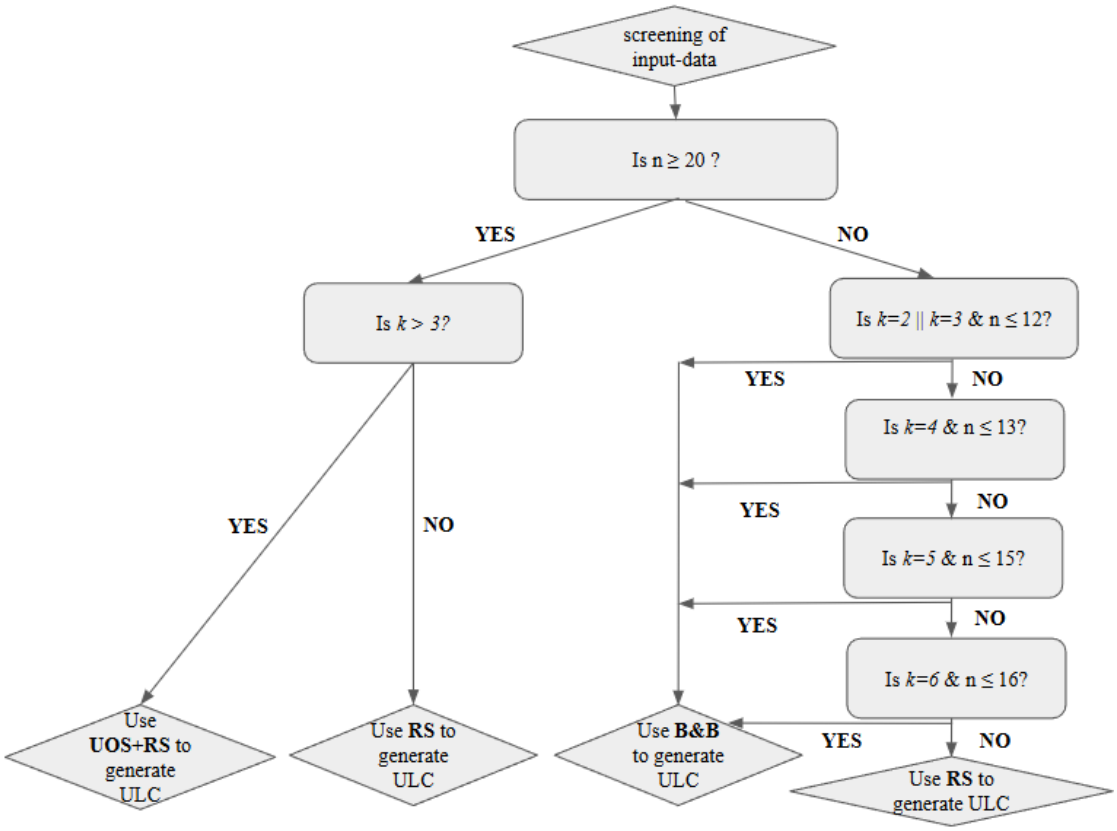

Figure 5.2: Decision tree describing the recommended procedure.

It can also be mentioned that the larger unique order-stacks consisting of 3- and 4-orders are not currently considered in the algorithm. As discussed in Section 3.9, larger UOS that consist of 3- or 4-orders were excluded due to the risk of violating the single-towering constraint and the hypothesis that forming larger stacks may reduce the total number of achievable UOS. However, for larger pick-orders, if there are sufficient orders with short box-accessibility spans to create UOS containing 3-orders, meaning $|uniqueOrderStacks_{3-orders}| \geq k$, then this would be a recommended step of future algorithm development.

5.2 Limitations and future research

The developed algorithm was based on a number of assumptions and constraints which are not considered in practice. In particular, the following assumptions were made:

i) All boxes were assumed to be of equal dimensions.

ii) Weight factors for PLCs and unit-loads were disregarded.

iii) Aisles were assumed to be equally dimensioned.

iv) Pick-orders were assumed to be static, and order due dates were ignored.

To improve the algorithm, stepwise relaxation of constraints would be a natural way of further enhancing its applicability. It can also be mentioned that in reality, the bin packing problem (optimally arranging boxes of varying dimensions into a unit-load to maximize space utilization) must be considered to some degree to allow boxes of different sizes to be stacked on top of each other in a stable way. Alternatively, the customer is influenced to only use a limited range of tote box sizes during picking. Solving the bin packing problem would most likely reduce the number of feasible box-stacking configurations, as differences in box dimensions make it more challenging in identifying stable solutions (for instance, larger boxes cannot be stacked on significantly smaller boxes). If the bin packing problem is considered as part of the algorithm, a decision must be made whether pallet packing density or traversal distance should be prioritized as the primary optimization objective.

A limitation of the algorithm is the fact that it could only be tested on identified pick-orders which contained up to 14 orders and 3 layers, while in practice, pick-orders can contain up to 42 orders in 6 layers. This means that the algorithm unfortunately has not been evaluated on larger actual cases, which is preferred and would be a required step before advancing to further refinement in the implementation phase. To simulate larger pick-orders, characteristics of the identified pick-order data was used to generate pick-orders that resembled real pick-orders.

For the generated cases containing pick-orders with high number of orders, such as cases 10 and 11, the improvement relative to the serpentine traversal is less than for the cases of the identified pick-orders. As discussed, this is probably due to the fact that items were stochastically allocated in the warehouse, which increased item dispersion. To further improve item generation and testing of the algorithm, analyzing the observed item locations for the identified pick-orders by either fitting another distribution or developing a heuristic framework is recommended. Such a heuristic, used for generating items for larger pick-orders, could consider factors such as clustering of orders, commonly avoided aisles and tendency for items belonging to the same order to be located within the same or adjacent aisles. This approach naturally relies on the assumption that the item locations for larger pick-orders follow the same allocation patterns as in cases 1 to 6. By creating pick-order cases that more closely resembles the item pick orders in cases 1 to 6, testing of the algorithm can be done more reliably, which is especially significant when the available input-data offer few test cases.

In this thesis, a pure random search algorithm is recommended as the ULC-generation approach for larger pick-orders with unit-load layers being partially filled with unique-order stacks. However, even though RS led to improvement over the baseline in all cases, it can never guarantee to find the optimal picking route. Therefore, an aspect to further research is to identify what other search algorithms could be used for successful ULC-generation. A natural step to improve RS would be to consider previous solutions. Ding, Fragkos and De Koster (2018) used improvement operators such as swapping boxes between layers or moving boxes to another layer as a second step in their algorithm to further decrease the traversed picking distance. These local search adjustments could similarly be integrated with RS to identify children solutions in the neighborhood of the current best ULC and thereby guiding the search toward better solutions.

Another search method that could be explored for the ULC-generation in the algorithm is a genetic algorithm (GA). The GA generates a pool of candidate solutions, selects the best candidates and iteratively evolves the population through selection, crossover and mutation operators (Katoch, Chauhan and Kumar, 2021). In the context of this thesis project, crossover would then refer to exchanging layer assignments of two ULCs, and mutation to randomly reassign pick-load carriers to different layers. Although genetic algorithms do not guarantee to find the optimal solution, they are often shown to provide solutions of good quality.

6. References

De Koster, R., Le-Duc, T., Roodbergen, K.J, (2007). European Journal of Operational Research, 182(2), 481–501. *Design and control of warehouse order picking: A literature review*.

Tompkins, J. A., White, J. A., Bozer, Y. A., & Tanchoco, J. M. A. (2010). Facilities Planning (4th ed.). John Wiley & Sons

Bartholdi, J. J., & Hackman, S. T. (2019). *Warehouse & Distribution Science* (Release 0.98.1). The Supply Chain & Logistics Institute, Georgia Institute of Technology.

Reid, H. (2024, November 8). *Cluster picking: A comprehensive guide*. DCL Logistics. Available at: https://dclcorp.com/blog/inventory/cluster-picking/ (Accessed: 28 March 2026).

Korf, R. E. (2003). *Search techniques*. In Encyclopedia of Artificial Intelligence (2nd ed.). Elsevier. Available at: https://doi.org/10.1016/B0-12-227240-4/00155-6

Allabolag. (2024). Industri-Matematik International AB – Bokslut & nyckeltal. Available at: https://www.allabolag.se

Affärsvärlden. (1997, 19 March). *INDUSTRI-MATEMATIK: Enda svenska nischspelaren*. Available at: https://www.affarsvarlden.se/artikel/industri-matematik-enda-svenska-nischspelaren-6743480)

Industri-Matematik International (IMI). (2025, 16 December). *Optimization for manual or automated picking. Or both?* White paper, Available at: https://im.se/insights/optimization-for-manual-or-automated-picking-or-both/ (Accessed: 9 March 2026).

Höst, M., Regnell, B., & Runeson, P. (2006). Att genomföra examensarbete. Lund: Studentlitteratur.

Petersen, C.G. (1997) *An evaluation of order picking routing policies*, International Journal of Operations & Production Management, 17(11), pp. 1096–1111. Available at: doi:10.1108/01443579710177860.

Roodbergen, K.-J., 2001. *Layout and Routing Methods for Warehouses* (No. EPS-2001-004-

LIS). ERIM Ph.D. Series Research in Management.

Aboelfotoh, M., Singh, N., & Suer, G. (2019). *Order batching optimization for warehouses with cluster-picking*. *Procedia Manufacturing*, 39, 1464–1473.
Available at: https://doi.org/10.1016/j.promfg.2019.07.136

Engati. (n.d.). *Brute-force search*.
Available at: https://www.engati.ai/glossary/brute-force-search (Accessed: 19 March 2026).

Zabinsky, Z. B. (2011). *Random search algorithms*. In Wiley Encyclopedia of Operations Research and Management Science. Wiley.
Available at: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470400531.eorms0704

Bergstra, J., & Bengio, Y. (2012). *Random search for hyper-parameter optimization*. Journal of Machine Learning Research, 13, 281–305.
Available at: https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf

Huang, C.-Y., Lai, C.-Y., & Cheng, K.-T. (2010). *Fundamentals of algorithms* (Chapter 4). In Facilities Planning (4th ed.). Wiley.
Available at: https://www.sciencedirect.com/science/chapter/edited-volume/pii/B9780123743640500114

Roodbergen, K. J., & De Koster, R. (2009). *Routing methods for warehouses with multiple cross aisles*. International Journal of Production Research, 39(9), 1865–1883.
Available at: https://doi.org/10.1080/00207540110028128

Ding, T., Fragkos, I., & De Koster, R. (2018). *Picker Routing and Pallet Stacking with Item Precedence Constraints*. Department of Industrial Engineering, Tsinghua University, and Rotterdam School of Management, Erasmus University.
Available at: https://www.sciencedirect.com/science/article/pii/S0360835218302869?via%3Dihub

Žulj, I., Glock, C. H., Grosse, E. H., Schneider, M. (2018) *Picker routing and storage-assignment strategies for precedence-constrained order picking*. Computers & Industrial Engineering, 123, 339–347.

Hillier, F. S., & Lieberman, G. J. (2001). *Introduction to Operations Research* (7th ed.). McGraw-Hill.

Katoch, S., Chauhan, S.S. and Kumar, V. (2021) *A review on genetic algorithm: past, present, and future*, Multimedia Tools and Applications, 80(5), pp. 8091–8126.
Available at: https://pmc.ncbi.nlm.nih.gov/articles/PMC7599983/

A. Appendix

A.1 Guide for interview on October 7, 2025, with Application consultant

Context
Introduction of myself and stating the purpose of this interview, which is to gain an understanding of the picking process, characteristics of pick-orders, warehouse layout and routing rules.

Introductory questions

- Would you mind giving a brief introduction and explaining your current role and responsibilities?
- How long have you been an application consultant?
- How much experience do you have with the WMS and software development?

Main questions

- How many aisles and shelves per aisle are there in the warehouse?
- Are unit-loads always of EU-pallet type? If not, what are some other common material handling types?
- How many pick-load carriers are usually put on a unit-load in total and how many of them are usually in a unit-load layer? How much can these parameters vary?
- How many items are typically picked to a pick-load carrier?
- Are items that have be picked to a pick-load carrier usually dispersed in the warehouse or are they stored collectively?
- Do order-pickers strictly follow the S-shaped pick-path, or do they eventually take shortcuts independently?
- Can there be more than one order-picker at a time in the path? If so, then how many at max?
- What pick-related data is stored in the WMS? How is the data reached in the source code?

A.2 Pseudocode for brute-force algorithm

Algorithm 4 Brute-Force search for optimal Unit-Load Configuration

function BRUTEFORCESEARCH(O_{all}, k)
  $D_{best} \leftarrow \infty$
  $candidates \leftarrow$ empty list of ULC_{cand}
  Generate all valid ULC_{cand} of O_{all} into layers of size k by GETALLPARTITIONS(O_{all}, k, $candidates$)
  for each ULC_{cand} in the generated set **do**
    $D_{tot} \leftarrow 0$
    for each layer $\in ULC_{cand}$ **do**
      $d_{layer} \leftarrow$ FINDSHORTESTPATH($n_{start}\emptyset$, n_{target}, 0)
      $D_{tot} = D_{tot} + d_{layer}$
    end for
    if $D_{tot} < D_{best}$ **then**
      $D_{best} \leftarrow D_{tot}$
      $ULC_{best} \leftarrow ULC_{cand}$ ▷ Current ULC with minimal total distance
    end if
  end for
  return ULC_{best}, D_{best}
end function

function GETALLPARTITIONS(O_{all}, k, $candidates$) ▷ Returns a set of all possible ULC-combinations
  if O_{all} is empty **then**
    return empty partition ▷ Base case — one valid complete partition found
  end if
  Create ULC_{cand} containing O_{all}
  for each k-combination $C \in O_{all}$ **do** ▷ Unique order comb. of size k from O_{all}
    Remove orders in $C \in O_{all}$ to form O_{rem}
    for each group in GETALLPARTITIONS($O_{remaining}$, k, $candidates$) **do** ▷ Recursive call
      Put C as a new layer and add the completed ULC_{cand} to $candidates$
    end for
  end for
  return $candidates$
end function

A.3 Pseudocode for branch-and-bound algorithm

Algorithm 5 Branch-and-bound search for optimal Unit-Load Configuration

function BRANCHANDBOUND(O_{all}, k)
  $D_{best} \leftarrow \infty$
  Calculate $d_{min} \leftarrow$ COMPUTEMINLAYERCOST(O_{all}, k) ▷ Lower bound on any single layer
  SEARCH([],0, $\emptyset$, O_{all})
  return ULC_{best}, D_{best}
end function

function COMPUTEMINLAYERCOST(O_{all}, k) ▷ Returns the min. cost of all possible layers
  $d_{min} \leftarrow \infty$
  for each k-combination $C \in O_{all}$ **do**
    $d_C \leftarrow$ FINDSHORTESTPATH(n_{start}, $\emptyset$, n_{target}, 0)
    if $d_C < d_{min}$ **then**
      $d_{min} \leftarrow d_C$
    end if
  end for
  return d_{min}
end function

function SEARCH(L_{curr}, D_{curr}, O_{used}, O_{all})
  if $|O_{used}| = |O_{all}|$ **then** ▷ Complete solution found
    if $D_{curr} < D_{best}$ **then**
      $D_{best} \leftarrow D_{curr}$ ▷ Update the shortest distance
      $ULC_{best} \leftarrow$ copy of L_{curr}
    end if
    return
  end if
  $O_{rem} \leftarrow O_{all} \setminus O_{used}$
  $r \leftarrow \lceil |O_{rem}|/k \rceil$ ▷ Remaining layers needed
  $lb \leftarrow D_{curr} + r \cdot d_{min}$ ▷ Update lower bound on total cost
  if $lb \geq D_{best}$ **then** ▷ Bound: prune this branch
    return
  end if
  for each $\min(k, |O_{rem}|)$-combination $C \in O_{rem}$ **do** ▷ Branch over next layer
    $d_C \leftarrow$ FINDSHORTESTPATH(n_{start}, $\emptyset$, n_{target}, 0)
    $D_{new} \leftarrow D_{curr} + d_C$
    if $D_{new} \geq D_{best}$ **then** ▷ Bound: prune
      continue
    end if
    Add C to L_{curr}
    Add all orders in C to O_{used}
    SEARCH(L_{curr}, D_{new}, O_{used}, O_{all}) ▷ Recurse
    Remove C from L_{curr}
    Remove all orders in C from O_{used} ▷ Backtrack
  end for
end function

A.4 Results from identified and generated cases

Table A.1: Results for identified and generated cases with 2 orders per layer. Traversal distances/costs are measured in meters and all values are averages over ten runs. Cases 7 to 11 represent the generated cases.

Case No.	BF. cost	B&B. cost	RS. cost	UOS+RS. cost	Serp. cost	BF. time [s]	B&B. time	RS. time
1	344	344	344	740	1480	0.14	1.03 s	1 min 11 s
2	608	608	608	899	1850	0.6	1.4 s	2 min 15 s
3	627	627	627	1122	2220	2.1	1.7 s	1 min 40 s
4	-	643	647	1127	2220	-	4 min 12 s	1 min 29 s
5	-	-	797	1272	2590	-	-	1 min 54 s
6	-	-	799	1272	2590	-	-	1 min 50 s
7	-	-	1763	1811	2960	-	-	1 min 30 s
8	-	-	1824	1829	2960	-	-	3 min 17 s
9	-	-	2017	2073	3700	-	-	2 min 8 s
10	-	-	2607	2848	5550	-	-	3 min 20 s
11	-	-	3369	3948	7770	-	-	9 min 6 s

Table A.2: Results for identified and generated cases with 3 orders per layer. Traversal distances/costs are measured in meters and all values are averages over ten runs. Cases 7 to 11 represent the generated cases.

Case No.	BF. cost	B&B. cost	RS. cost	UOS+RS cost	Serp. cost	BF. time [s]	B&B. time	RS. time
1	319	319	319	524	1110	0,3	1,98	1 min
2	499	499	499	669	1480	2,96	41	1 min 32 s
3	508	508	508	754	1480	30	18 s	1 min 39 s
4	-	540	540	857	1480	-	9 min 32 s	1 min 40 s
5	-	-	637	902	1850	-	-	2 min 16 s
6	-	-	664	902	1850	-	-	2 min 13 s
7	-	-	1309	1319	1850	-	-	1 min 27 s
8	-	-	1397	1441	2220	-	-	1 min 5 s
9	-	-	1768	1804	2590	-	-	2 min 20 s
10	-	-	2160	2159	3700	-	-	2 min 33 s
11	-	-	2876	3016	5180	-	-	4 min 11 s

Table A.3: Results for identified and generated cases with 4 orders per layer. Traversal distances/costs are measured in meters and all values are averages over ten runs. Cases 7 to 11 represent the generated cases.

Case No.	BF. cost	B&B. cost	RS. cost	UOS+RS. cost	Serp. cost	BF. time [s]	B&B. time	RS. time
1	267	267	267	370	740	0,21	0,33 s	49 s
2	445	445	445	529	1110	1,3	7,53 s	1 min 12 s
3	413	413	413	622	1110	6,9	43 s	1 min 17 s
4	458	458	487	691	1110	0,94	1 min 10 s	56 s
5	-	565	578	727	1480	-	3 min 25 s	1 min 46 s
6	-	-	593	694	1480	-	-	1 min 48 s
7	-	-	1200	1205	1480	-	-	1 min 1 s
8	-	-	1274	1289	1480	-	-	57,8 s
9	-	-	1473	1408	1850	-	-	42 s
10	-	-	1858	1806	2960	-	-	1 min 10 s
11	-	-	2328	2131	4070	-	-	3 min 23,5 s

Table A.4: Results for identified and generated cases with 5 orders per layer. Traversal distances/costs are measured in meters and all values are averages over ten runs. Cases 7 to 11 represent the generated cases.

Case No.	BF. cost	B&B. cost	RS. cost	UOS+RS. cost	Serp. cost	BF. time [s]	B&B. time	RS. time
1	264	264	264	370	740	0,37	0,29	51,74 s
2	390	390	390	370	740	0,56	0,88	43,68 s
3	418	418	418	502	1110	3,62	9,38	77,6 s
4	452	452	453	585	1110	15,97	36,89	69,05 s
5	-	510	526	622	1110	-	5 min 11 s	1 min 14 s
6	-	514	538	622	1110	-	20 min 26 s	1 min 14 s
7	-	960	988,5	1006,5	1110	-	23 min 11 s	25 s
8	-	-	1066	1124	1480	-	-	1 min 10 s
9	-	-	1280	1238	1480	-	-	1 min
10	-	-	1625	1525	2220	-	-	1 min 10 s
11	-	-	2276	2177	3330	-	-	3 min 6,07 s

Table A.5: Results for identified and generated cases with 6 orders per layer. Traversal distances/costs are measured in meters and all values are averages over ten runs. Cases 7 to 11 represent the generated cases.

Case No.	BF. cost	B&B. cost	RS. cost	UOS+RS. cost	Serp. cost	BF. time [s]	B&B. time	RS. time
1	255	255	255	370	740	0.28	0.24 s	27.57 s
2	390	390	390	370	740	0.94	1.07 s	35.52 s
3	390	390	390	484	740	1.92	2.44 s	40.24 s
4	406	406	406	535	740	1.42	3.12 s	31.17 s
5	448	448	480	532	1110	39.80	31.63 s	23.0 s
6	-	456	480	532	1110	-	3 min 33 s	43,4 s
7	-	932	952	996	1110	-	16 min 58 s	31.2 s
8	-	910	962	1050	1110	-	17 min 8 s	28.2 s
9	-	-	1140	1147	1480	-	-	30.81 s
10	-	-	1601	1328	1850	-	-	49.7
11	-	-	1965	1816	2590	-	-	1 min 27 s

A.5 Case 5 with visualized bar diagram and ULC

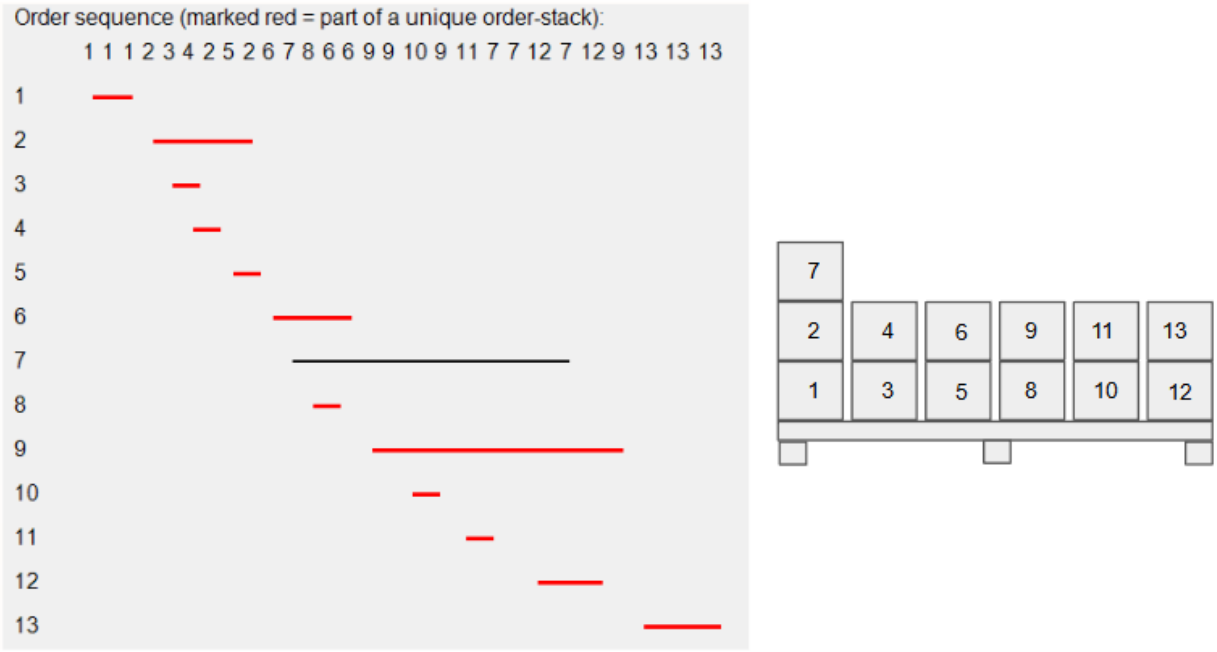

Figure A.1: Bar diagram and visualization of the best ULC found in case 5.