UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**

Exploring the World of Container Stacking using Approximate Dynamic Programming

René Boschma
Master Thesis

Supervisors:
prof. dr. ir. H.J. Broersma
dr. ir. M.R.K. Mes
L.R. de Vries MSc

Formal Methods and Tools
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Contents

Abstract

Intermodal shipping containers will be, at one point in their transport, stored at a terminal. In this terminal, containers are usually stacked on top of one another. This approach yields a higher utilization of the area available but leads to unproductive moves when a lower stacked container needs to be retrieved. Such a move is commonly referred to as a reshuffle and preventing them has a financial benefit as it not only avoids the costs of executing the reshuffle but also decreases the time needed to retrieve a container. Most researches consider the problem where all containers need to be removed from a terminal using the least number of reshuffles possible [19]. This problem is commonly referred to as the *Container Relocation Problem* and is proven to be NP-Hard [3]. It only considers the departure of containers and, usually, it is assumed that the departure time of containers is fully known in advance. In reality, this is often not the case [11]. In addition, no research addresses the stacking restrictions imposed by a reach-stacker, which is most often used in smaller inland terminals. Therefore, in this thesis, a stacking approach, which tries to minimize the number of reshuffles, is proposed. It considers both arrival and departures of containers and includes uncertainty in arrival and departure times. Furthermore, it includes restrictions imposed by the use of reach-stackers.

The problem tackled in this thesis is a sequential decision problem that includes uncertainty. Therefore, the problem is modeled using Stochastic Dynamic Programming (SDP). As the number of possibilities to store containers grows exponentially, it is decided to focus in this thesis on Approximate Dynamic Programming (ADP). In this thesis, the data of seven terminals are analyzed. From this analysis, assumptions are drawn from which a model is defined. Different ADP approaches are tested on smaller problem instances. The best-found approach is tested on real-life instances and compared to existing approaches in literature. The real-life instances are generated using data found in the analysis of the terminals.

Results show that a basis function approach using the least squares method to update its weight performs best of the tested ADP approaches on the smaller instances. The decision space still proved to be too big to yield results in a reasonable time for real-life instances. Therefore, the decision space was split into smaller decisions and the basis function was used to guide the search of the smaller decisions. This method is referred to as the optimized basis function approach. Still, the optimized basis function proved not to be suitable for real-life instances. In addition to the optimized basis function, a new method is proposed. This method limits the allowed decisions using a corridor. All decisions making use of resources outside the corridor are ignored. This method is referred to as the corridor method and proved to yield compelling results for real-life instances in a reasonable time. It was tested against different terminal sizes where the biggest contained 1000 stacks. It outperformed the compared heuristic found in literature significantly. All data and code used can be found at: https://github.com/boschma2702/ContainerStacking.

Chapter 1

Introduction

Intermodal containers are used to transport various goods efficiently across different modes of transport. Containers exist in various shapes and sizes. Commonly the term *Twenty foot Equivalent Unit* (TEU) is used to express size in terms of area of a container, vessel and terminal. One TEU is equivalent to the area of a 20-foot container. In 2017 the port of Shanghai, the highest-ranking port in terms of container throughput, had a throughput of 40 million TEU [32]. This is an increase of 8.3 percent compared to 2016. Worldwide the annual container throughput increased from 560 million TEU in 2015 to 793 million TEU in 2017 [31]. The increasing throughput demands a higher capacity of ports.

Two main container handling moves can be identified: *loading* and *unloading*. During loading, containers are moved from a carrier (for instance a vessel, train or truck) to a terminal. During unloading the reverse path is followed (from terminal to carrier). From a practical point, the terms loading and unloading mean something different than in literature. Here loading and unloading are viewed from the point of the carrier. For example, one loads a vessel rather than a terminal. Thus, the loading problem (in literature) concerns the unloading of carriers. Therefore, in this thesis, loading and unloading are referred to as the handling of in- and outbound containers.

To increase the capacity and minimize the area needed, containers are stacked on top of one another. Stacking containers impose a Last-In-First-Out (LIFO) retrieving order within a stack of containers. If one wants to retrieve a container, first all containers above need to be moved to a different stack. Such a move is called a *reshuffle* and is often regarded as an unproductive move [12]. Preventing reshuffles has a financial benefit as it not only avoids the costs of executing the reshuffle but also decreases the time needed to retrieve a container. Fewer reshuffles during unloading, thus, decreases the time needed to retrieve all containers for a carrier. This results in higher throughput for the terminal and less time wasted waiting for the carrier. In literature, the problem of preventing reshuffles is known as the *Container Relocation Problem* (CRP) and *Blocks Relocation problem* (BRP)).

One can reshuffle containers prior to the handling of outbound containers (for example, when no vessels are at the port). This process is commonly referred to as pre-marshaling and aims to decreases the number of reshuffles during the handling of outbound containers. One could reduce the need for pre-marshaling by incorporating an approach that minimizes reshuffles. Such an approach would seek stack assignments, during the handling of inbound containers or

reshuffling, that minimizes the number of reshuffles. In smaller to mid-size terminals, such stack assignments are usually done by hand and may not guarantee the most efficient assignments in terms of minimizing the number of reshuffles. In the remainder of the thesis, the term efficient use of storage space refers to minimizing the number of reshuffles.

Different problem flavors of efficient stack assignments exist in literature. One can consider the handling of inbound containers, outbound containers, pre-marshaling moves, and combinations thereof. In this research, the focus is on the handling of in- and out-bound containers. Various approaches to this problem exist in literature. Li et al. [20] use a simulation integrated with a genetic algorithm to tackle the combined problem. Ries et al. [26] use a fuzzy logic model to describe the inbound problem. Wan et al. [34] describe an integer programming model and a heuristic for the outbound and combined problem. Galle et al. [9] tackles the outbound problem using a binary integer programming model. Güven and Türsel Eliiyi [13] propose two heuristics to the combined problem. Zeng et al. [36] describe five heuristics for the outbound problem. Kim and Hong [15] propose a branch and bound algorithm that can solve smaller instances for the outbound problem optimally and a heuristic that can solve bigger instances. Tanaka and Takii [28] describe 3 lower bound functions and use these to optimize a branch and bound algorithm for the outbound problem.

This research tackles the combined problem. It is sought to bring theoretical research and practical use closer together. Various researches ([4, 13, 15]) assume that the arrival and departure times are known, while in reality the exact arrival and departure times seem to be unpredictable [11]. This unpredictability makes it rather difficult or even impossible to control the order in which containers arrive and depart [12]. Researches that regard uncertainty are limited to either the loading or the unloading problem ([12, 17, 38] and [10, 18]). The combined problem is a sequential decision problem and therefore lends itself to be modeled using dynamic programming. As uncertainty is included, the problem will be modeled using stochastic dynamic programming (SDP). Because real-life settings are too big to solve exactly using modern-day computers, approximations need to be used. Using approximations in DP is also also known as Approximate Dynamic Programming (ADP). This yields the following research question:

> *"How to determine real-life applicable container stack allocations that minimizes the number of reshuffles?"*

In order to answer this question, first, the following sub-questions are answered:

1. Which stacking approaches, concerning the handling of in- and outbound containers, are known in literature?
2. What are arrival, departure and stacking characteristics of shipping containers?
3. How to model the stacking problem?
4. How to apply ADP to the described model?
5. How well does the found ADP solution compares to other approaches?

1.1 Approach

This research is divided into three phases: *Research Phase*, *Implementing Phase* and *Evaluation Phase*. In the research phase, the first two research questions are answered. Related stacking approaches are identified and real-life data is analyzed to get insights into the practical

perspective of the problem. In the second phase, the third and fourth research questions are answered. The SDP model is defined and different ADP approaches are described and compared. In the third and last phase, the best ADP approach found will be applied to real-life settings and compared to approaches found in the first phase. The three phases will be executed sequentially. An overview of the phases and corresponding research questions is shown in Figure 1.1. This figure also shows which research question is answered in which chapter.

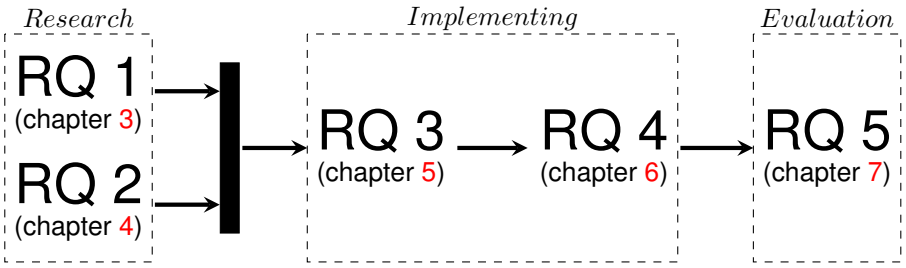

Figure 1.1: Figure showing the three phases of the research and the research questions answered during these phases.

1.2 Structure

The remainder of the thesis is structured as follows: Chapter 2 gives some practical background of the problem and discusses pre-marshaling and the handling of in- and out-bound containers in more detail. Chapter 3 identifies and discusses known approaches to the stack allocation of containers during the handling of in- and out-bound containers. Chapter 4 analyzes data from seven terminals. Chapter 5 first gives an introduction into SDP. Afterward, it elaborates on the made assumptions as well as describing the proposed model. Chapter 6 first gives an introduction into ADP. Then various ADP approaches are applied to smaller problem instances and results are discussed. Chapter 7 compares the best found ADP solution with existing approaches. Comparison is done first on smaller instances and later on real-life instances. Chapter 8 will end with a conclusion.

Chapter 2

Background

This research is carried out in the name of Cofano[1]. Cofano is a software development company located in Enschede and Sliedrecht. They offer a Software as a Service (SaaS) solution which aids customers in the management of container transportation. It, for instance, keeps track of when containers arrive and depart, but also where they currently are. The remainder of this chapter discusses some background knowledge to introduce readers unfamiliar to the domain. First, basic information on intermodal containers is given. Second, equipment used to handle containers is described. Third, problems known in the literature are discussed. Fourth and lastly, the scope of the research is defined.

2.1 Containers

Intermodal containers come in different sizes and types. The two most common sizes are 20-foot and 40-foot. The 40-foot is twice the size, in terms of area, of the 20-foot. Figure 2.1 shows the exact sizes of both containers. Besides the area, there also exist containers with the same area but different heights.

Besides various sizes, also various types of containers exist. The variations mentioned in this section cover only a small number of possibilities. As there are too many variations to cover, the interested reader is referred to the ISO-64346 standard [14]. This standard describes all standardized variations of containers. The most noteworthy is the reefer. A reefer is a container that has active cooling and thus needs to be connected to electricity when stored in a terminal. Figure 2.2 shows a reefer.

Containers are usually stored on fixed pre-determined spots called *stacks*. Containers can be placed on top of each other. To indicate the height of a stack, the term *tier* is used. Stacks are commonly placed side by side. A group of such side by side placed stacks is referred to as a *bay*. In turn, bays are grouped to form a block[2]. Figure 2.3 shows a single block with three bays. Each bay consists of four stacks.

[1] https://www.cofano.nl/

[2] In literature, the term block is sometimes used as a synonym for a container (for instance, in the block relocation problem). In this thesis, the term block will only be used to refer to a block as shown in 2.3 (i.e. multiple bays grouped).

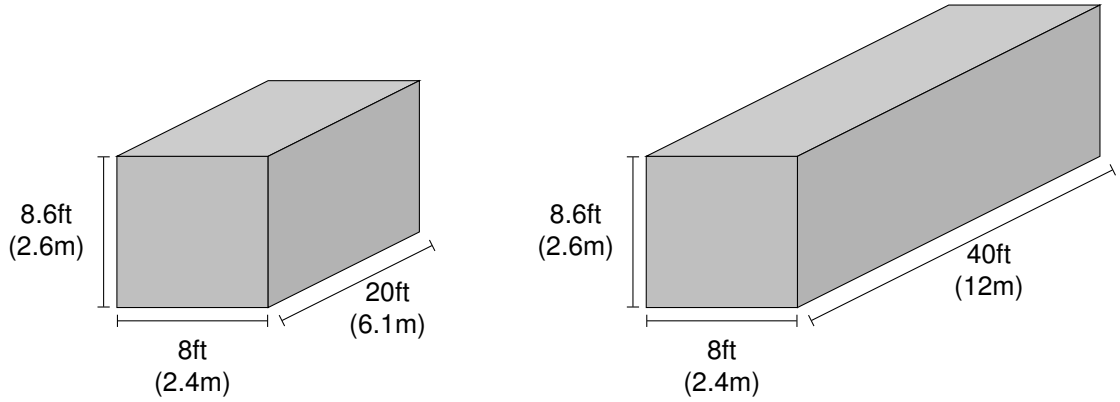

Figure 2.1: Dimensions of both a 20-foot (left) and a 40-foot (right) container.

Figure 2.2: Image showing a reefer container [33]

2.1.1 Container Transport

Two main flows of container transport can be identified: *import* and *export* [7]. Import containers are containers that arrive in large vessels from overseas and continue their destination through inland transport. Inland transport can be via inland shipping, train and truck. Export containers arrive from inland and leave via vessels overseas. In addition, a third category can be identified: *transshipment containers* [16]. These containers are loaded from a vessel, temporarily stored and then placed back onto a different vessel. Arrival times of import containers are often somewhat predictable compared to export containers [7]. On the other hand, departure times of export containers are more predictable than those of import containers.

2.1.2 Stacking Restrictions

Some containers impose restrictions on where they can be placed or what kind of container can be placed on it. For instance, one can mix 20-foot and 40-foot containers in a single stack. A 20-foot can be stacked on a 40-foot but not vice versa. It is, however, possible to stack a 40-foot on top of two 20-foot containers.

Besides size restrictions, the type of the container or its contents might impose restrictions. For

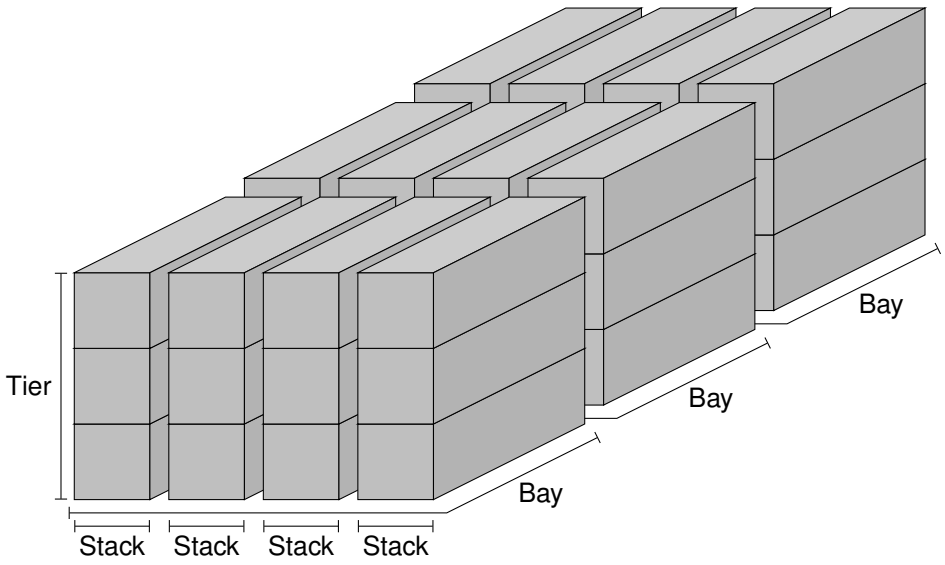

Figure 2.3: Figure showing a block. A block consists of multiple bays which consist out of multiple stacks and tiers.

instance, when filled, a reefer needs to be connected to a power supply. Another example is a container that contains dangerous goods. Depending on the substance, it may not be allowed to place it near other containers. The Economic Commission for Europe Committee on Inland Transport [8] defines nine classes of dangerous goods. Each class has its restrictions on where a container may or may not be stacked.

2.2 Equipment

There exists a wide variety of equipment for handling containers [30]. In stacking, the two most used pieces of equipment are the gantry crane and reach-stacker. Each has its advantages and disadvantages. A gantry crane can retrieve all top-most containers, whereas a reach-stacker has more complicated rules on which containers it can reach. It can only retrieve and store containers directly in front of it or a row behind when the height is one higher than the stack in front of the reach-stacker. On the other hand, a gantry crane is tied to a specific part of the terminal, whereas a reach-stacker is free to roam. Figure 2.4 shows both a gantry crane (left) and reach-stacker (right).

(a) A Gantry Crane [6] (b) Reach Stacker [5]

Figure 2.4: Figure showing left a gantry crane and right a quay crane

2.3 Problems

The introduction briefly discussed the three main problems present during the stacking of containers; *handling of inbound containers*, *handling of outbound containers* and *pre-marshaling*. In this section, those three problems are explained in more detail.

2.3.1 Handling Inbound Containers (Loading Problem)

Handling inbound containers is the situation where one is given a sequence of arriving containers and tries to minimize the number of reshuffles when retrieving all containers in order of departure. An example is shown in Figure 2.5. In this example, a non-empty terminal layout is given and the arrival sequence is $1, 5, 2, 4$. The numbers denote the priority of the containers (i.e. a lower number indicates an earlier departure time). A solution that does not results in a reshuffle is shown to the right of the arrow (note that in this example there exists only one such solution). Besides a sequence, also a set can be used as an arrival sequence. In this case, the precedents among items are not relevant. This gives more flexibility in assigning a container to a good spot. An example where containers arrive in such a manner is when a vessel arrives. Following a stowage plan, one is free to pick the next container to retrieve from the vessel.

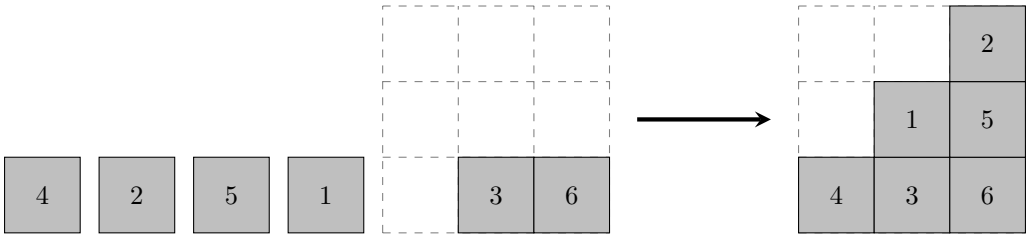

Figure 2.5: Figure showing a solution of the given stacks and the arrival order of $1, 5, 2, 4$ that does not result in a reshuffle. The number indicates the priority of a container.

2.3.2 Handling Outbound Containers (Unloading Problem)

Handling outbound containers (also known as the Block Relocation Problem (BRP) or Container Relocation Problem (CRP)) concerns the situation where one is given a storage layout filled with items. Each item has a priority or departure time and must be retrieved in that order. An item can only be retrieved when there is no container above it. If there is an item above the one that needs to be retrieved, this item is called blocking. First, all blocking containers must be moved to a different stack (reshuffled), before the item can be retrieved. The goal is to empty the storage using as few reshuffles as possible. Finding an optimal solution is NP-hard [2]. An example is given in Figure 2.6. In this figure, the number indicates the priority of a container. As can be seen, at least two reshuffles are needed to retrieve all containers.

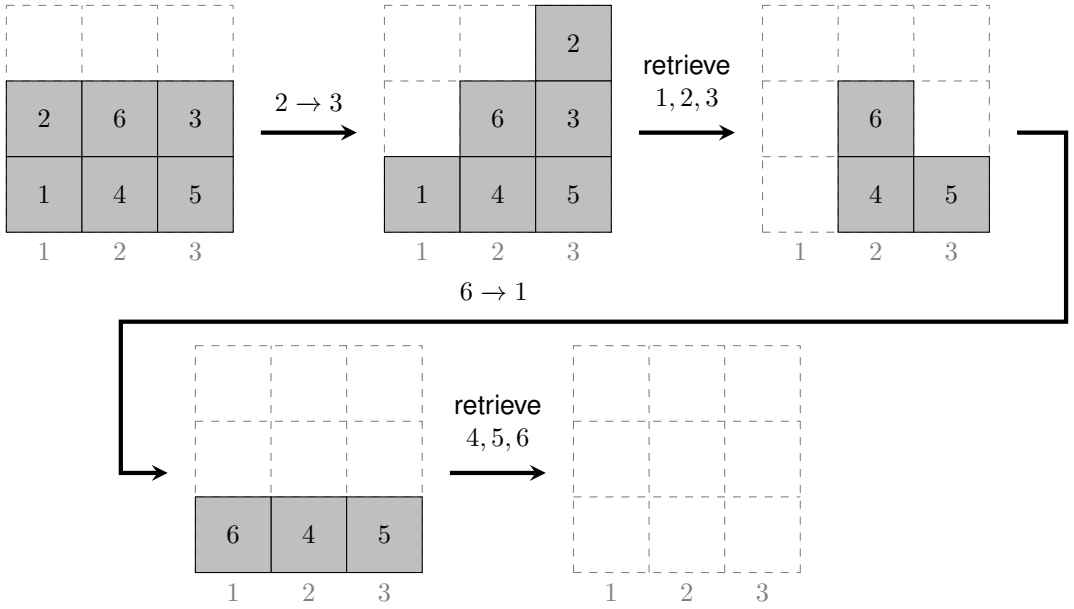

Figure 2.6: Example that shows the handling of outbound containers. At least two reshuffles are needed in order to retrieve all containers.

2.3.3 Premarshalling

Besides the handling of inbound and outbound containers, one can also move containers when no containers arrive and depart. Such moves are done in a post-loading period (i.e. when no vessel is at the port). The pre-marshaling problem tries to minimize the future moves while minimizing the total number of moves during the pre-marshaling process itself. The pre-marshaling problem is comparable to the unloading problem, only no containers leave the storage. Figure 2.7 shows an example of the pre-marshaling problem. In this example, multiple items have the same priority and gray and white are used to distinguish between the individual containers. The optimal solution can be found using 4 reshuffle moves.

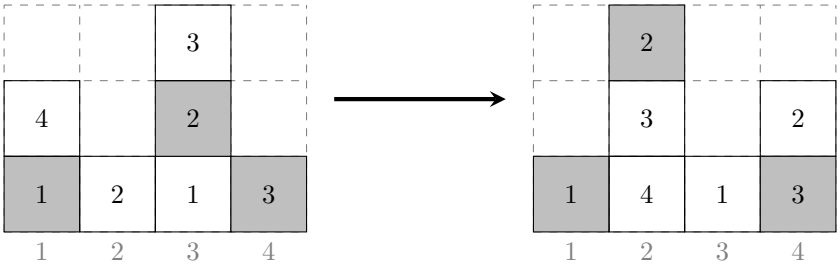

Figure 2.7: The Figure on the left shows the initial bay layout and the figure on the right shows the bay layout after pre-marshaling. Gray and white are used to distinguish between containers of the same priority. A solution which does not cause a reshuffle can be achieved in 4 moves: white 2 to stack 4, white 4 to stack 2, white 3 to stack 2 and gray 2 to stack 2.

2.4 Research Scope

Terminal management can be seen as a three-level problem [7]. The first level is the *strategic* level. Strategic concerns the choices of equipment. This includes which machines to use, but also the layout of the stacks (i.e., where each stack is located in a terminal). The second level is the *tactical* level. Tactical decisions concern the capacity decisions on months to a year. These decisions include the number of cranes and ground vehicles deployed, but also whether to use pre-marshaling. The third level is the *operational* level. Operational deals with decisions on where to put a container on a given moment of time and assignments of equipment to jobs.

This research will focus on the operational side of the problem. It is assumed that the stack layout (i.e., where containers can be placed in a terminal) is given, sufficient equipment is present and disregard deployment thereof. To limit the complexity of the problem, pre-marshaling is also eliminated. From a practical point of view, it makes sense to first focus on the handling of both in- and out-bound containers before taking pre-marshaling into account. Efficient handling may diminish the effect and need for pre-marshaling. An overview of the scope, based on the problem levels described and identified by Dekker et al. [7], can be found in Figure 2.8. In this figure, each problem regarded in the proposed research is colored gray.

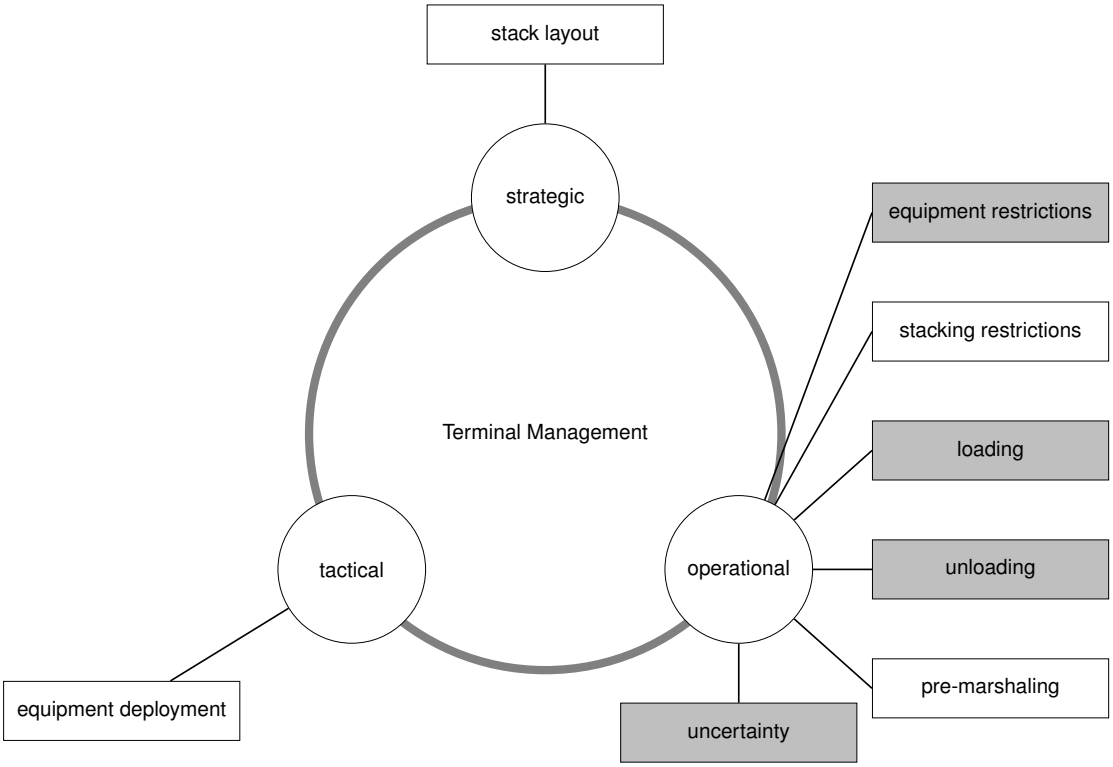

Figure 2.8: An overview of various identified problems occurring in yard management. Each level is indicated with a circle and each problem is indicated using a rectangle. When a problem is colored gray, it is regarded in this research.

2.5 Conclusions

This section explained that there exists a wide variety of containers. It also introduced basic concepts such as *stack*, *bay*, *tier* and *block*. Furthermore, it gave a brief explanation on the transport flow of containers as well as stacking restrictions that are imposed by the type of container. Then it explained the two main kinds of equipment used for stacking. Moreover, this section, gave a more detailed description of the loading, unloading and premarshalling problem as found in literature. Lastly, this chapter defined the scope within the broad domain of terminal management.

The concepts discussed in this section serve as a foundation for the remainder of the thesis. Throughout this thesis, the concepts described in this chapter will be used. It is assumed that the reader is now familiar with these concepts. The next chapter will seek to answer the first research question by analyzing literature.

Chapter 3

Related Work

This section seeks to answer the first research question: *"Which stacking approaches, concerning the handling of in- and out-bound containers, are known in literature?"*. Different variations of the handling of in- and out-bound containers are described in literature. To distinguish the different variations, the classification scheme introduced by Lehnfeld and Knust [19] is used. This chapter first elaborates on the classification scheme. Second, various stacking approaches are discussed.

3.1 Classification Scheme

Lehnfeld and Knust [19] propose a classification schema that applies to the handling of in- and out-bound containers. It also applies to the pre-marshaling problem and combinations with the handling of in- and out-bound containers. The classification scheme forms an abstraction on the most common problems considered in the literature. The schema consists out of three fields: $\alpha|\beta|\gamma$. Each field is explained in detail below.

The α-field

The α field consists of three parameters: α_1, α_2 and α_3. α_1 describes the problem considered.

$$\alpha_1 = \begin{cases} L, & \text{if handling of inbound items is considered} \\ U, & \text{if handling of outbound items is considered} \\ P, & \text{if the pre-marshaling problem is considered} \\ LU, & \text{if a combination of handling in- and out-bound items is considered} \\ LP, & \text{if a combination of handling inbound items and pre-marshaling is considered} \end{cases}$$

α_2 and α_3 specify the parameters of the storage area.

$$\alpha_2 = \begin{cases} m = m', & \text{if the number of stacks is equal to a fixed number } m' \in \mathbb{N} \\ m \geq n, & \text{if the number of stacks is larger or equal to the number of items} \\ \circ, & \text{if the number of stacks } m \text{ is given as part of the input} \end{cases}$$

and

$$\alpha_3 = \begin{cases} b = b', & \text{if the limit b on the number of items in a stack is given by a fixed number } b' \in \mathbb{N} \\ b = \infty, & \text{if no limit on the number of items in a stack is given} \\ \circ, & \text{if the limit } b \text{ is given as part of the input} \end{cases}$$

The β-field

The β field describes whether in- and out-bound items are specified as either a sequence, a set, a sequence of sets or a set of sequences.

For the loading (L) problems are used:

$$\beta_1 = \begin{cases} \pi^{in}, & \text{if a sequence of inbound items is given} \\ \mathcal{I}^{in} & \text{if a set of inbound items is given} \\ (\mathcal{I}^{in})_K, & \text{if a sequence of length } K \text{ containing sets of inbound items is given} \\ \{\pi^{in}\}_K, & \text{if a set of } K \text{ sequences of inbound items is given} \end{cases}$$

For the unloading (U) and premarshaling (P) problems are used:

$$\beta_1 = \begin{cases} \pi^{out}, & \text{if a sequence of outbound items is given} \\ \mathcal{I}^{out} & \text{if a set of outbound items is given} \\ (\mathcal{I}^{out})_K, & \text{if a sequence of length } K \text{ containing sets of outbound items is given} \\ \{\pi^{out}\}_K, & \text{if a set of } K \text{ sequences of outbound items is given} \end{cases}$$

For the combined handling of in- and out-bound items (LU and LP) we use:

$$\beta_1 = \begin{cases} (\pi^{in}, \pi^{out}), & \text{if first one sequence of inbound items and then a set of outbound items is given} \\ ((\mathcal{I}^{in})_K, (\mathcal{I}^{out})_{K'}), & \text{if first one sequence of length } K \text{ containing sets of inbound items and then one sequence of length } K' \text{ containing sets of outbound items is given} \\ (\mathcal{I}^{in}, \mathcal{I}^{out})_K, & \text{if a sequence of length } K \text{ containing sets of inbound and outbound items is given} \end{cases}$$

In addition to Lehnfeld and Knust [19] we also define $(\pi^{in}, \pi^{out})_K$. This case denotes K sequences of in- and out-bound items. It specifies the situation where both handling of in- and out-bound items happens in a fixed (intertwined) sequence. It can be noted that $(\mathcal{I}^{in}, \mathcal{I}^{out})_K$ covers this case by only allowing sets of size one and zero for both $\mathcal{I}^{in}$ and $\mathcal{I}^{out}$[1]. In our opinion it is nice to have a specific definition for this case as the broader definition may be used to cover a different problem.

During the handling items, one might choose from a group of items instead of serving a specific one. An example of this would be empty containers. These can be interchanged as long as the owner of the container is the same. In that case, we say that these containers belong to the same family. A a subscript $\mathcal{F}$ is added when families of items are handled. For instance, $\pi^{out}_{\mathcal{F}}$ then denotes a sequence of families that need to be retrieved instead of specific items. If

[1] $((1,2)^{in}, (2,1)^{out}, (5,7)^{in}, (5)^{out}) \equiv (\{1\}^{in}, \{\}^{out}, \{2\}^{in}, \{2\}^{out}, \{\}^{in}, \{1\}^{out}, \{5\}^{in}, \{\}^{out}, \{7\}^{in}, \{5\}^{out})$

retrieval times are unknown and estimated by past statistics a $\sim$ is used, thus $\tilde{\pi}^{out}$ denotes an unknown retrieval sequence based on statistical data.

β_2 specifies which stacking restrictions need to be respected.

$$\beta_2 = \begin{cases} S_{ij}, & \text{if stacking restrictions have to be respected} \\ \circ, & \text{if stacking restrictions do not have to be respected} \end{cases}$$

Here S_{ij} is defined as a 2-dimensional matrix ($S \in \{0,1\}^{n \times n}$ where a 1 means that item i is stackable on top of item j. A 0 indicates item i is not stackable on top of item j.

Besides the limit on the number of items in a stack (defined by the α_3-field), one can also define a weight and height limit.

$$\beta_3 = \begin{cases} weight\_limit, & \text{if the total weight of each stack is limited} \\ \circ, & \text{otherwise} \end{cases}$$

and

$$\beta_4 = \begin{cases} height\_limit, & \text{if the total height of each stack is limited} \\ \circ, & \text{otherwise} \end{cases}$$

Some items may not be placed at certain spots. Take for instance reefers, these need to be placed on spots where electricity is available.

$$\beta_5 = \begin{cases} L_i, & \text{if location restrictions for items are given} \\ \circ, & \text{otherwise} \end{cases}$$

Here L_i denotes the possible locations an item i may be stacked.

Two types of moves can be identified: *forced* and *voluntary*. A forced move is a move that moves an item that is blocking (i.e. has an item below it that needs to be retrieved first). Voluntary moves also allow moving items that are not blocking. This means that arbitrary items can be moved.

$$\beta_6 = \begin{cases} v\_moves, & \text{if voluntary moves are allowed} \\ \circ, & \text{otherwise} \end{cases}$$

After a move, items can be put back onto their original stack or remain at the new designated location. If an item must be put back on its original position, this is called a push-back.

$$\beta_7 = \begin{cases} push\_back, & \text{if items have to be pushed back} \\ \circ, & \text{otherwise} \end{cases}$$

The γ-field

The γ-field indicates which objective function is used. A list of possible objective functions is given below.

- $-$: Decide whether a feasible solution for the given problem exists.
- $\#RS$: Minimize the number of reshuffles.

- $\#RS = 0$: Decide whether the problem can be solved without any reshuffles.
- $\mathbb{E}(\#RS)$: Minimize the expected number of reshuffles. This is applicable in situations where the retrieval times are unknown but can be estimated by a probability distribution.
- TC: Minimize transportation costs. Transportation costs are mostly the time a crane handles a container.
- $\#US$: Minimize the number of unordered stackings with respect to the retrieval order given. One can count the unordered stackings in two ways. The first *adjacent* is denoted as US_{adj}. This method counts the number of two adjacent items where the upper item blocks the lower one. This method of counting yields a lowerbound for the CRP. The second, *depth* denoted by US_{dep}, counts the number of containers above the lowest blocked item in a stack. This method of counting yields a lowerbound for the premarshaling problem. Figure 3.1 illustrates the difference between the two counting methods.
- $\mathbb{E}(\#US_{adj}(\tilde{w}))$: Minimize the number of expected unordered stackings with respect to estimated weights $\tilde{w}$. This objective is used in situation where the weight of inbound containers is uncertain.
- f: Minimize a given objective function f.

The introduced functions can be considered independently or in combination. For instance $w_1\#RS + w_2TC$ denotes the weighted sum of the number of reshuffles and transportation costs with weights $w_1, w_2 \geq 0$. More details on the classification schema can be found in [19].

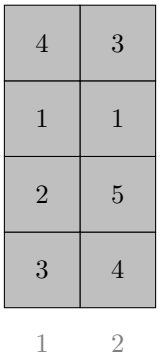

Figure 3.1: Figure depicting the difference between US_{dep} and US_{adj}. Stack 1: $US_{adj} = US_{dep} = 1$. Stack 2: $US_{adj} = 2$, $US_{dep} = 3$. In stack 2 item 4 is the lowest blocked item in the stack and has 3 items above it.

3.2 Stacking Approaches

Various approaches have been applied to the different problem instances of the stacking problem. This section describes in more detail the identified stacking approaches. The approaches are divided into two main categories *deterministic approaches* and *stochastic approaches*. A table that summarizes all identified approaches can be found in Section 3.2.4. Besides approaches, also measures that indicate how well a terminal is stacked are addressed in literature. After describing both the deterministic and stochastic approaches, these measures are discussed.

3.2.1 Deterministic Approaches

The deterministic approaches assume that all times are known in advance. In other words, no uncertainty is present in the order of arrival and/or departure. Many pieces of research make this assumption. Therefore only a brief overview of some of the different approaches is discussed. Each section below discusses works grouped based on the approach used. Some approaches only discuss a single work.

Genetic Algorithm

A genetic algorithm is a meta-heuristic based on the process of natural selection. It evolves around chromosomes that influence which decisions are made. Chromosomes are evaluated using simulations and mixed and mutated to converge to a solution that yields good performance for the targeted problem. Li et al. [20] use a genetic algorithm to tackle the handling of inbound containers. Besides trying to minimize the number of reshuffles, it also tries to minimize the travel distance of containers. A chromosome consists of all the stack allocation for the inbound containers.

Fuzzy Logic

Fuzzy logic is a form of multi-valued logic where truth values of variables may be a real number ranging from 0 to 1. Ries et al. [26] propose a fuzzy logic model for handling inbound containers. Besides trying to minimize the number of reshuffles, it also tries to minimize the travel distance of containers. A stack is chosen by first picking a block. The block is chosen based on the block utilization and the distance between the block and the exit of the terminal. Within a block, the stack is chosen based on the stack heights and the estimated time of departure of the top container.

Integer Programming

In integer programming (IP), one defines a set of equations where variables are restricted to be integers. Caserta et al. [1], Galle et al. [9], Tang et al. [29] and Wan et al. [34] use IP to tackle the handling of outbound containers. Wan et al. and Tang et al. also address the combined problem. This is achieved by taking the outbound problem and regarding inbound containers as containers that need to be reshuffled. As an IP model takes a long time to solve and is impractical for real-life instances, Wan et al. and Tang et al. propose various heuristics.

Caserta et al. and Galle et al. present an IP model where all stacks are represented using a single matrix. This matrix offers the possibility to extract the stack location of a container efficiently. It also provides an efficient way to see which containers are located above a certain container. Block relocations and reshuffles are defined as matrix manipulations.

Heuristic

A heuristic is an approach to problem-solving that embodies a practical method that is not guaranteed to be optimal but is sufficient for reaching the intended goal. Güven and Türsel Eliiyi [13] propose a heuristics for the inbound problem. The location for a stack is chosen based on the departure time, size, trade type and weight. Also, the receiver, owner and destined vessel is taken into account. Zeng et al. propose five heuristics for the outbound problem. In the described model, container departures are based on the group arrival information of external

trucks. This information is based on how the appointments in a Truck Appointment System (TAS) work. In such a system, trucks are given time slots in which they can bring or retrieve a container. From these time slots, one can group containers. Within a group, the order of serving does not matter. Order among groups does matter. The proposed heuristics are adjusted from [3] and [29].

Branch and Bound

Branch and Bound (B&B) is a tree search algorithm that explores branches of a tree. Before a branch is explored, the lower- and upper-bound of that branch are determined. The branch is disregarded when the bounds are worse than the currently best-found solution. Both Kim and Hong [15] and Tanaka and Takii [28] use B&B to solve the outbound problem optimally. Kim and Hong uses the number of blocking containers as a lower bound, while Tanaka and Takii uses a more sophisticated lower bound function based on lower bounds described in [15] and [39]. As the B&B approach of Kim and Hong is not appropriate in practice, a heuristic is proposed.

Dynamic Programming

In Dynamic Programming (DP), one tries to cut the problem into smaller, more manageable, problems in a recursive manner. Caserta et al. [4] applies DP to the outbound problem. In the proposed method (the Corridor Method (CM)), not all stacks are evaluated. Only stacks within a certain distance of the original stacks are evaluated (the horizontal corridor). Besides this horizontal corridor, also a vertical corridor can be included. This way only items lower than a certain threshold are evaluated. These corridors limit the state space and are a way to deal with the "Curse of Dimensionality" arising in DP approaches. From a physical point of view, it makes sense to introduce such corridors. Stacks in terminals have limited height and it does not make sense to move a container from one outer point of a yard to another outer point. The proposed method was tested on randomly generated bay layouts where each stack has the same height and compared to the approach of Kim and Hong [15]. The proposed solution was found to perform better, although taking more time. The times for larger instances were still reasonable to prove usability in real-life applications. The CM could empty 10000 containers from a 100 wide and 102 high bay with corridor sizes (both horizontal and vertical) of two in under 5 minutes. It needed 20000 fewer reshuffles than the approach of Kim and Hong (roughly 87000 reshuffles were needed for the CM).

3.2.2 Stochastic Approaches

Stochastic approaches include uncertainty. Arrival and/or departure times are not fully known in advance and decisions are made based on incomplete information. To the best of our knowledge, all researches including uncertainty are included in this section. Most stochastic researches define an SDP. Therefore, division in this section is made based on the method of solving the underlying model, rather than the modeling itself. Two methods were identified: *Decision Trees* and *Approximate Dynamic Programming*.

Decision Tree

A Decision Tree (DT) uses a tree-like structure to model decisions and their possible consequences, including chance outcomes and costs. Galle et al. [10], Gharehgozli et al. [12], Kim et al. [17] and Ku and Arthanari [18] use a decision tree to solve their proposed SDP model.

Gharehgozli et al. and Kim et al. use the decision tree to tackle the stochastic variant of handling inbound containers whereas Ku and Arthanari and Galle et al. tackle the stochastic variant of handling outbound containers.

In the SDP model of Gharehgozli et al., containers are divided into families based on the ship it is to be loaded on, its weight and its port destination. It is assumed that containers of family f are to be retrieved before family $f+1$ and arrive uniformly. Stacks are distinguished by the number of free spots and the family of the stack. The family of a stack is defined as the family of the first to departure container in the stack (note that this does not has to be the topmost container). It is assumed that reshuffled containers do not generate new reshuffles. The goal is to minimize the number of expected reshuffles. Computation of solving the SDP model exactly takes to long for bigger instances. Therefore a decision tree heuristic is proposed. The proposed solution uses the results of SDP applied to small-scale instances and uses it to generate generalized decision trees that can solve large-scale problems.

Kim et al. divide containers into weight groups (heavy, medium and light). They assume a container with a heavier weight group is to be retrieved before lighter ones. A state in this model is represented by the number of empty slots in a stack and the highest weight group in the stack. It is assumed that the weight class of the next container to arrive is uniformly distributed. Solving the SDP model exactly takes too long to solve for bigger instances. Therefore it creates a decision tree based on the exact solution of smaller instances to solve bigger ones.

The model of Ku and Arthanari is based on truck retrievals. Trucks are appointed to a certain time interval. The precedents among truck arrivals that fall within the same time interval are unknown. The information on which truck arrives next (and thus which container needs to be retrieved next) is assumed to be uniform (i.e. every truck within the same time interval has the same probability of arriving next). Information on the k^{th} container to be retrieved is known when the $(k-1)^{th}$ container is retrieved. The described problem is referred to as the *CRP with Time Windows* (CRPTW). To solve the SDP model, a search-based algorithm is proposed. The search-based algorithm models the SDP as a decision tree with chance nodes. A decision node depicts an actual decision made (for instance, moving a container). A chance node represents a possible outcome of a stochastic process (i.e. the arrival of a truck). The decision tree is searched using depth-first search (DFS) with backtracking to save memory consumption. To reduce the size of the search space, abstraction is used. A stack representation is transformed into an abstract one by removing all empty stacks and reordering the columns in ascending order of the pickup priorities from the bottom tier to the top tier. Although the abstraction can improve the time performance several times, the exact calculation still does not have reasonable computational times to be used in real-life settings. Therefore, an index-based heuristic is proposed that can be applied to the CRPTW.

Galle et al. introduces the *batch model*. In this model, like the CRPTW, the order of outbound containers is not fully known in advance. It introduces the notion of batches, which containers belong to. The order among batches is known, but ordering within a batch not. The difference between both models is that the CRPTW model reveals the next container to depart per container retrieval, the batch model does this per batch (i.e. the order within a batch is revealed before a batch is handled). If one is currently handling batch $k \in \{1, \ldots, W\}$ then the order of all containers within batch $k' > k$ is not yet known. After handling the last container in batch k, the order of batch $k+1$ is revealed. From this model, a DT is created. Two search methods are proposed: Pruning Best-First Search (PBFS) and PBFS-approximate (PBFSA). PBFS considers all possible retrieval orders within a batch and calculates the number of reshuffles needed to optimally retrieve the batch. It does this bread-first by first considering states with the lowest

lower-bound and disregard states that have a lower lower-bound than the current best solution. PBFSA works the same as PBFS, only it considers a subset of the possible outcomes.

Approximate Dynamic Programming

Zhang et al. [38] extends the work of Kim et al. [17] and Zhang et al. [37]. It proposes two adaptations to the original model. The first adaptation is a new way of calculating the punishment coefficient (value function) of the SDP model. The proposed adaptation takes the magnitude of a decision into account. For instance, placing a heavy container on an empty stack increases the likelihood of a lighter one being placed on a heavier one. The second adaptation changes the representation of the state. The old model had no information on which weight groups were present in the group, only the heaviest was known. In the new model, information on which groups are present in a stack is included. Both new models are compared to the original one and found to yield better solutions. As the SDP model with the second adaptation can only solve smaller instances exactly, a rollout ADP algorithm is proposed. In the rollout algorithm, the value function is replaced by an approximation. The approximation is determined by simulating multiple arriving sequences and solving this using some base policy. This yields simulated values of the value function. The average of these simulated values is used as an approximation for the value function. The base policy used uses a priority list of stacking locations generated from the adapted reward function. The rollout algorithm did not outperform the DT approach of Kim et al. [17] but is still promising when a good base policy can be found.

3.2.3 Feature Functions

Besides approaches, literature describes various features that try to give some insights into how well a given stack layout is. Some features introduce a lower-bound on the problem, while others solve it, yielding an upper-bound. This section only discusses measures to the CRP, as no relevant measures for the combined problem could be identified. First, three lower-bounds measures are discussed, after which four upper-bounds. Lastly, an approximate measure based on the lower- and upper-bounds is described. All measures are applied to the stack layout displayed in Figure 3.2. It is possible to empty this layout using 6 reshuffle moves.

Lower Bound

Blocking Number ($\#blocks$) The blocking number ($\#blocks$), introduced by Kim and Hong [15], compares all pairs of items stacked directly above one another. $\#blocks$ counts the number of containers which have a container directly below itself that departs earlier. Figure 3.2 shows an example where the containers that contribute to $\#blocks$ are colored gray. As can be seen, $\#blocks = 3$ in this example.

Number of Confirmed Reshuffles ($\#CR$) The number of confirmed reshuffles ($\#CR$) counts the number of items that have an item that departs earlier below it. It can be noted that $\#CR \geq \#blocks$. Figure 3.2 shows an example where the containers that contribute towards $\#CR$ are horizontally striped. As can be seen, $\#CR = 4$ in this example.

Number of Confirmed Reshuffles with Look-Ahead ($\#CRL$) The number of confirmed reshuffles with look-ahead ($\#CRL$), introduced by Zhu et al. [39], takes consequences of a

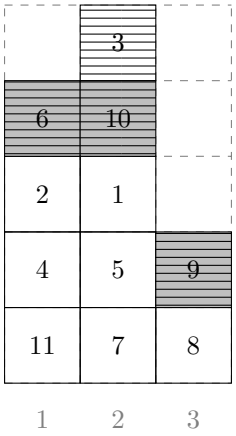

Figure 3.2: Figure depicting the counted containers in $\#blocks$ and $\#CR$. Items colored gray are counted for $\#blocks$ and items horizontally striped are counted for $\#CR$.

reshuffle into account. One counts the normal reshuffles and the guaranteed additional reshuffles to determine this lower bound. After a reshuffle, the reshuffled container is disregarded in the remainder of the computation. For instance, when looking at Figure 3.2, first container 3 needs to be reshuffled. For this container, a position exist that causes no additional reshuffles (stack 3). Next, container 10 needs to be reshuffles. Both options result in a future reshuffle, thus this reshuffle yields a guaranteed additional reshuffle. The remainder of reshuffles do not yield a guaranteed reshuffle. In total there are 4 normal reshuffles and 1 additional, resulting in $\#CRL = 5$.

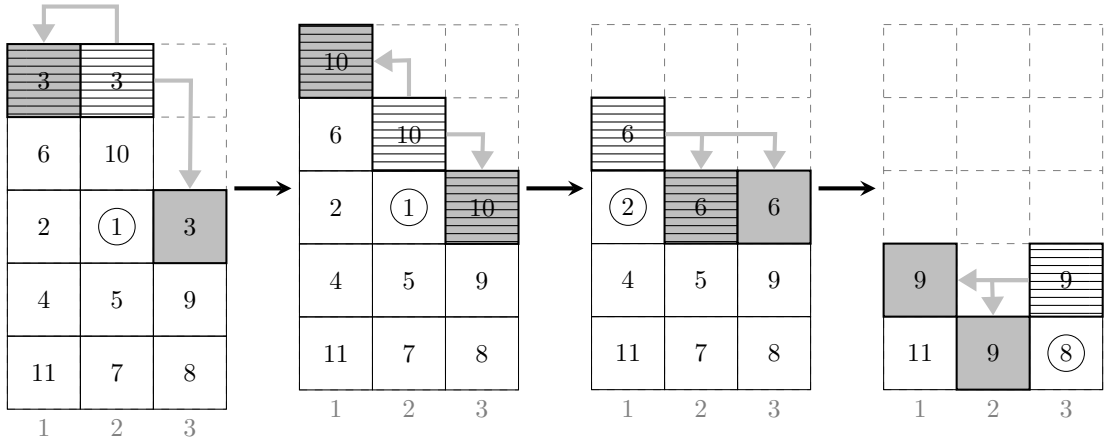

Figure 3.3: Figure illustrating the approach to determine $\#CRL$. A circle indicates the current retrieving item. Gray items indicate the possible reshuffle locations. Horizontal stripes indicate if an item is blocking.

Upper Bound

Reshuffle Index (RI-$rule$) This rule, introduced by Murty et al. [22], relocates a blocking container c to the stack with the lowest reshuffle index (RI). Ties are broken by choosing the leftmost stack. The reshuffle index for stack s is defined as the number of items within a stack

with an earlier departure time than the one being reshuffled. Let $H(s)$ be the current number of containers in stack s and $c_1, \ldots, c_{H(s)}$ be the containers in stack s (where c_1 is the lowest container), then the RI is defined as:

$$RI(s,c) = \sum_{i=1}^{H(s)} \mathbb{1}\{c_k < c\} \tag{3.1}$$

Applied to the layout depicted in Figure 3.2, following the $RI\text{-}rule$ leads to 9 reshuffles.

Ku and Arthanari [18] adjust the $RI\text{-}rule$ so it can be applied to the stochastic CRP and refer to it as the expected reshuffle index (*ERI*). Rather than calculating the number of reshuffles a container c causes, it now calculates the expected number of reshuffles. Furthermore, rather than comparing the departure times of the containers it now compares the batch numbers. A smaller batch number indicates an earlier departure time, while the same batch number indicates uncertainty in which departs earlier. ERI is defined as:

$$ERI(s,c) = \sum_{i=1}^{H_s} \mathbb{1}\{c_i < c\} + \mathbb{1}\{c_i = c\}/2 \tag{3.2}$$

Reshuffle Index with Look-Ahead ($RIL\text{-}rule$) This rule, applied by Wu and Ting [35], extends the $RI\text{-}rule$ by changing the decision when multiple stacks have the reshuffle index. The earliest departing container of each stack is compared and the one that has the latest departing container is chosen. The idea behind this approach is that by delaying the additional reshuffle, it might happen that in the meantime another stack is freed (thus not introducing a new reshuffle). Applied to the layout depicted in Figure 3.2 it yields 8 reshuffles.

Min-Max Heuristic ($MM\text{-}rule$) This rule, introduced by Caserta et al. [1] and Caserta et al. [3] first chooses a stack where the first departing container, departs later than the current container (i.e. a stack that causes no additional reshuffle). This choice disregards empty stacks. If multiple such stacks exist, the stack containing the earliest departing item is chosen. In case no such stacks exist and no empty stacks are present, the $MM\text{-}rule$ chooses the stack that has the latest earlier departing item (similar to the $RIL\text{-}rule$).

Let $min(s)$ be the earliest departing container in stack s (if s is empty, then $min(s) = C + 1$ where C is the number of containers in the initial layout). Let c be a container from stack s that needs to be reshuffled and S be the set of all stacks. Then the reshuffle location s' for c is defined as:

$$s' = \begin{cases} \text{argmin}_{i \in S \setminus \{s\}} \{min(i) : min(i) > c\}, & \text{if } \exists i : min(i) > c \\ \text{argmax}_{i \in S \setminus \{s\}} \{min(i)\}, & \text{otherwise} \end{cases} \tag{3.3}$$

Applied to the layout depicted in Figure 3.2, the $MM\text{-}rule$ yields 7 reshuffles.

Galle et al. [10] adapts the $MM\text{-}rule$ so it can be applied to the stochastic CRP and refer to it as the expected min-max (*EM*). Just as ERI, it compares batch numbers rather than the departure times of the containers. Furthermore, new tie breaks for both cases are introduced. For the case when $\exists i : min(i) > c$ the highest stack is chosen (remaining ties broken by choosing the left-most). Ties for the second case ($\forall i : min(i) \leq c$) are broken by selecting the stack that has the least number of containers with priority $\max\{\min(i)\}$. Ties are further broken by taking the highest one and then choosing the leftmost one.

Expected Group Assignment (EG) EG is proposed by Galle et al. [10] and is applied to the stochastic CRP. Like ERI and EM, it compares batch numbers rather than departure times of containers. It consists of two phases. In the first phase, it considers all blocking containers at the same time. Let us denote R as the set of containers above the current target container. We try to assign, for all containers $c \in R$ in stack s, a stack s' such that $s \neq s'$, s' not assigned a $c' \in R$ original above c in stack s and $min(s) > c$. This assignment is done by first assigning containers with the lowest priority breaking ties for the highest one first.

It might happen that containers in R remain that have not yet been assigned. In the second phase, these are assigned a stack. This time the containers with a higher priority are assigned first. Let $B(s')$ denote the subset of containers assigned to stack s' in the first phase, T the max height of a stack and $H(s')$ the height of stack s'. Then stack s' is picked that has the highest $Gmin$, breaking ties identical to the second case of EM. $Gmin$ is defined as:

$$Gmin(s') = \begin{cases} -1, & \text{if } |B(s')| + H(s') = T \\ min(s'), & \text{if } |B(s')| = 0 \\ B(s'), & \text{if } |B(s')| = 1 \\ 0, & \text{otherwise} \end{cases}$$

Note that this index is recalculated after each container assignment. The Gmin value can, more intuitively, be defined as: if a stack is full, it can not be selected. If no container was assigned in the first phase, then the index remains as defined originally. If one container was assigned, then this container forms the new minimum. Finally, if more than one container was assigned, the index becomes very unattractive by being as low as possible. The second phase is similar to the EM heuristic but uses Gmin instead of min. Applied to the layout depicted in Figure 3.2 it leads to 6 reshuffles.

Estimate

Composite Measure ($CM(x, \alpha)$) Scholl et al. [27] compares the results of the above-mentioned features to the deterministic CRP and introduces a new measure to estimate the number of reshuffles when using an optimal approach. Compared to the previous features, this method yields an estimate rather than a bound. The measure is referred to as $CM(x, \alpha)$. $CM(x, \alpha)$ is a linear combination between the $MM - rule$ and some lower-bound measure. It is defined as: $CM(x, \alpha) = \alpha * MM\text{-}rule + (1 - \alpha) * x$, where $x \in \{\#block, \#CR, \#CRL\}$. It is found that $CM(\#CRL, 0.65)$ gives the best predictions of the true value on the number of reshuffles. Applied to the layout depicted in Figure 3.2 it estimates 6.3 reshuffles.

3.2.4 Overview

This chapter discussed various works that tackles some instance of handling in- and/or outbound containers. An overview of all discussed works can be found in Table 3.1. The overview shows the different fields used in the classification of Lehnfeld and Knust [19]. Sometimes papers discuss multiple alternatives to the problem. When this occurs, the cell in the table is split into multiple rows. One must follow the row picked across the entire row of the table. For instance, Galle et al. [9] cover the problem of handling outbound containers with voluntary moves where the goal is to minimize the number of reshuffles. They, however, do not cover the problem of handling outbound containers with voluntary moves where the goal is a combination of minimizing the number of reshuffles and crane travel costs (CT).

Work	Problem	limits	Order	Constraint	Goal	Method
[20]	L		π^{in}		$\#RS + TC$	GA
[26]	LU		$(\tilde{\pi}^{in}, \tilde{\pi}^{out})_K$		$\#RS + TC$	FL
[34]	U		π^{in}		$\#RS$	IP, H
	LU		$(\mathcal{I}^{out})_K$			
[29]	U		π^{in}		$\#RS$	IP, H
	LU		$(\mathcal{I}^{out})_K$			
[9]	U		π^{out}		$\#RS$	BIP
					$\#RS + CT$	
		v_moves			$\#RS$	
[13]	U	$m = 3360$ $b = 4$	π^{in}	$w = 3ton$ S_{ij}[2]	$\#RS$	H
[36]	U		$(\mathcal{I}^{out})_K$		$\#RS$	H
[15]	U		π^{out}		$\#RS$	B&B, H
			$\pi^{out}_{\mathcal{F}}$			
[28]	U		π^{out}		$\#RS$	B&B
			$(\mathcal{I}^{out})_K$			
[12]	L		$\tilde{\pi}^{in}_{\mathcal{F}}$		$\mathbb{E}(\#RS)$	SDP, DT
[17]	L		π^{in}		$\mathbb{E}(\#US_{adj}(\tilde{w}))$	SDP, DT
[38]	L		π^{in}		$\mathbb{E}(\#US_{adj}(\tilde{w}))$	SDP, ADP
[4]	U		π^{out}		$\#RS$	DP
[18]	U		$\tilde{\pi}^{out}$		$\mathbb{E}(\#RS)$	DT, H
[10]	U		$\tilde{\pi}^{out}$		$\mathbb{E}(\#RS)$	DT, H

Table 3.1: Overview of found related work and the tackled problems. The used methods are also included: Genetic Algorithm (GA), Fuzzy Logic (FL), (Binary) Integer Programming ((B)IP), Heuristic (H), Branch & Bound (B&B), Stochastic Dynamic Programming (SDP), Decision Tree (DT), Approximate Dynamic Programming (ADP) and Dynamic Programming (DP)

3.3 Conclusions

This chapter sought to answer the question: *"Which stacking approaches, concerning the handling of in- and out-bound containers, are known in literature?"*. This was accomplished by first introducing the classification scheme of Lehnfeld and Knust [19]. Afterward, various works were identified and discussed. Each work was grouped based on the method used to solve the underlying problem. In addition, various measures, that give insight into how well a terminal is stacked, were described. This chapter closed by giving an overview of all found related work and classified using the described classification scheme.

This chapter has given a broad overview of the number of different problem variations and possible solving strategies. The next chapter will answer the second research question by analyzing real-life data.

[2] where S_{ij} is defined as follows: a stack may not contain a mix of TUE and FUE containers.

Chapter 4

Container Handling Characteristics

This chapter will answer the second research question:*"What are arrival, departure and stacking characteristics of shipping containers?"*. The goal of this question is to get insights into the day-to-day practices of real-life terminals. These insights will be used later in the research to create and verify the described model and solution. The data analyzed, originates from workers that manually log their actions, which may result in inaccuracies in the data. However, the data can still provide broad insights into the characteristics of real-life terminals.

The analyzed data originates from seven clients of Cofano. Clients are referred to as Clients A till G. From these clients, C and E belong to the same company. They are located at the same location and operate on the same terminal. However, in the system, they are registered as two separate companies as the containers they handle are distinct. Therefore it is chosen to regard them as distinct clients in the analysis.

Furthermore, the data marks two periods which do not reflect the normal in- & out-bound flow of containers. The first period is when the client first adopts the system. The second period is when the export took place. This export marks a point in time where the terminal did not handle the containers yet. As these periods do not reflect the normal flow, it is decided to filter out these points. The start date of each client marks the first day it handled 20 containers during a single day (arbitrary chosen). The stop date includes all containers that depart earlier than the latest container arrived. Alongside the start and endpoint filtering, containers with an unknown arrival or departure date are removed.

The remainder of this chapter discusses the analysis of the real-life terminals. The analysis is divided into three parts. The first part answers questions related to the size, in terms of the number of containers handled, of the terminal. This information will be used to determine the scale which the proposed solution should support. The second part focuses on the arrival and departure times of containers and includes the dwell time of containers. This data will be used to generate real-life like test data. The third and final part describes the currently used stacking practices as well as terminal layouts. The latter is used to recreate the terminals digitally and compare the current practices with the proposed solution. Each analysis is based on graphs supplied in the appendix. The graphs used for a specific section are mentioned in the section header.

4.1 Handled Containers

To get insights into the size of each client, first, the number of in- and out-bound containers are discussed. Afterwards, via which modality the containers arrive and depart is presented. Lastly, the distribution of various kinds of containers is examined.

Number of In- and Out-Bound *[Figures A.1 and A.2]* Three out of seven clients (B, C, D and F) do not handle more than 500 inbound containers per week. On average, Client B handles around 300, Client C and F around 250 and Client D around 150. Of these clients, a high variability is present at Clients D and F. Client F has two peaks around 800, whereas the peaks of Client D do not surpass 450. An interesting phenomenon at Client F is that one week they receive relatively a high number of containers, while the next week a relatively low number is received. A similar thing occurs at Client D, but here the peaks drop two to three times before restoring.

The other three clients are larger and handle on average roughly 1000 containers per week (A, E and G). Of these clients, E starts around 500 containers per week and grows up to 1500 per week. Furthermore, it has three peaks reaching close to 2000. After this, Client E drops back to an average of around 1000. Clients A and G are relatively stable in the number of containers handled.

The same holds for the outbound containers. The only major difference is that the peaks of client E are more extreme and frequent. Client E has 8 peaks above 1500 and 5 peaks reaching 2500 for the outbound case, whereas for the inbound case three peaks above 1500 and one reaching 2000.

In- and Out-Bound Modality *[Figures A.1 and A.2]* The system of Cofano identifies three modalities: truck, train and barge. Of these modalities, all clients use the truck. All but one client (B) use barge transport and two clients (B and D) use the train. The ratio of inbound handlings between truck and barge is the same at most of the clients (D, E, F and G). At two clients, the number of inbound handlings via truck is twice the number then via barge (A and C). This can be observed in Figure 4.1.

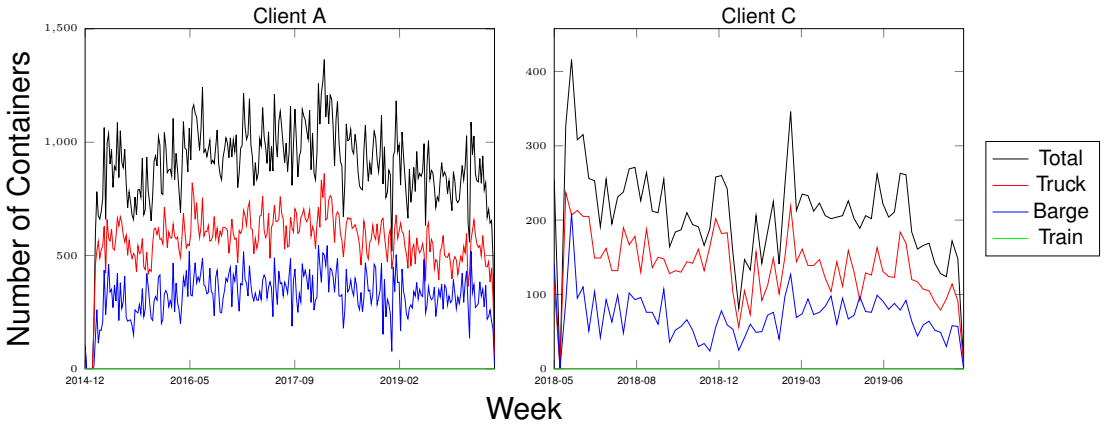

Figure 4.1: Overview of the number of inbound containers handled per week. The colour indicates via which modality the containers departs.

Client F has large peaks and drops in the number of containers handled by barge. These peaks can be explained by the fact that F is a deepsea terminal. This means that this client handles larger vessels less regular than clients that deal with inland vessels.

The same story applies to the outbound handlings. Only the two to one ratio of truck and barge is present at a single client (A).

Container Type ***[A.3, Figures A.4 and A.5]*** A distinction can be made based on the TEU of a container. Almost all containers handled are either one or two TEU. Some clients handle around the same number of one and two TEU (C, E and G), while others differ significantly (A, B, D, F). These clients almost solely handle a single TEU.

Another distinction can be made based on if a container is a reefer, an empty container or a non-empty (normal) container. When making this distinction, only three clients (A, B and F) handle reefers regularly. Of these clients, Client F almost solely handles reefers.

An interesting thing pops up when looking at the TEU size of reefers. Close to all are two TEU. When looking at the empty and non-empty containers, it can be noted that the ratio between them is the same.

Conclusions From the above analysis, the following is concluded. Clients vary in size. Three clients handle around, on average, 1000 containers per week. The remainder of the clients does not handle more than 500 per week. The smallest client handles around 150 containers per week. Most containers arrive and depart via trucks. All but one client uses barge transport and only two make use of train transport. The majority of the clients mostly handle a single TEU.

4.2 Arrival and Departure Times

To get insights into the arrival and departure times of containers, first, the weekday on which the client handles containers is discussed. Then the difference between the actual and expected arrival and departure times is discussed. Lastly, the dwell times are analyzed.

Weekday ***[Figure A.6]*** In general, clients only handle containers during weekdays. On an average weekday, the smallest client handles around 60 containers and the biggest 300 containers. During an average day, half of the containers handled are inbound while the other half are outbound. Overall the number of handled containers are the same over the weekdays. At three clients there is a significant difference between weekdays (C, G and F). For Clients C and G this is due to fewer barge transport on average on these days. Client F has both fewer barge and truck transports on average during these days.

Expected Arrival and Departure Times ***[Figures A.7 and A.8]*** The expected arrival and departure times are based on what is registered by the client. The client follows the following chain to determine the expected time: If the transport for the container is planned, this date is used. When the transport has not been planned, the date from when the container can be obtained or stored is used. If this is not known, the date when the customer wants the container is used. When all information is not present, the system does not set an expected arrival or departure time. Containers, for which no expected arrival and departure time are known, are

omitted in this analysis. Furthermore, sometimes the expected times are filled in after the actual arrival. In this case, the expected time is set to the actual time.

At five clients there is a large peak at zero (A, B, C, D and G). This peak might be explained by the fact that sometimes the expected times are filled in after the actual arrival, causing a zero difference in expected versus actual. The remainder of the difference in expected and actual arrival times are very variable and hard to take clear points from.

Dwell Time [Figure A.9] Generally, the dwell time follows an exponential distribution. This is most notable at Clients A and C. Figure 4.2 shows the dwell times of these two clients.

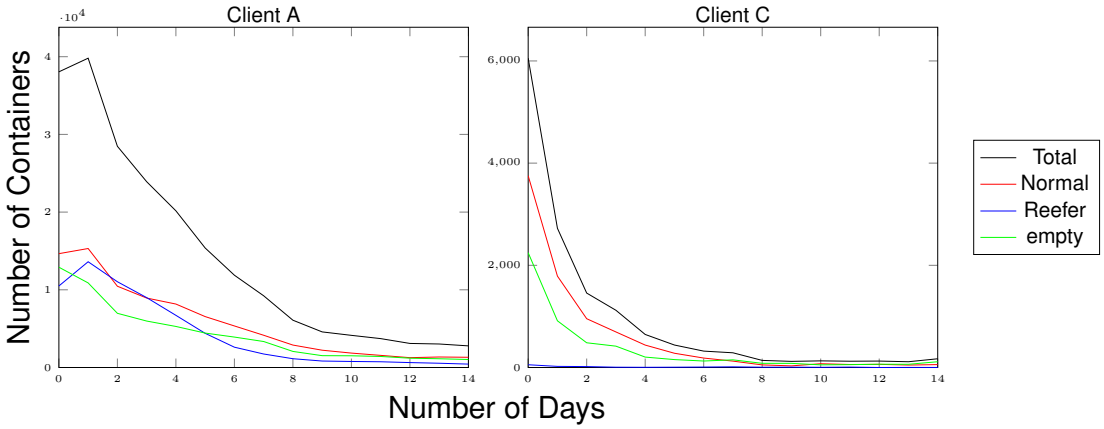

Figure 4.2: Overview of the number of days a container remains in the terminal for Clients A and C.

Two exceptions of the exponential dwell time distributions are Clients F and G. At Client F a small peak is present after seven days. The dwell time at Client G is quite unpredictable and does not follow an exponential distribution. The average dwell time ranges from two to four days per client and averages (non-weighted) around three days.

Conclusions From the above analysis, the following conclusions can be drawn. During the week, containers are solely handled during the weekdays. This means that in a seven day period, only five are used for handling containers. The data of the expected arrival and departure times are to random to extract meaningful information from. Lastly it is assumed that the dwell time of a container follows an exponential distribution with a mean of three days.

4.3 Stacking

Clients vary in the precision of recording where and when a container is placed. For instance, Client A has over half a million handlings but reported of only 4312 handlings the location. At client D, the container moves are not correctly reported. Either the move is not reported at all or the locations are not specified. Therefore Clients A and D are omitted for the remainder of this section.

The locations where a client can place a container each have an identifier. This identifier is a string specified by the client and, therefore, it is unknown how the naming scheme works. For

instance, at some clients, the naming of locations seems to resemble the structure of bay-stack-tier. Here tier ranges from 1 till 4 for all places. Others resemble the structure of block-bay-stack, where stack ranges from 1 till 14. Replaying the in- and out-bound containers using the reported times and counting the number of containers that resides at a given moment at a location confirms these ideas. For a block-stack-tier combination, no more than a single container occupied the location at a given moment, while the bay-block-stack usually only holds at most 4. However, names like square and track[1] are used, which hint towards a general location where containers are not stacked. For this analysis, the effort has been made to normalize all locations to, what is presumed, a single stack. As the number of general locations is small, it is chosen to count these as individual stacks for the sake of simplicity.

The remainder of this section discusses, first, the number of stacks per client. Then the homogeneity of the stacks is analyzed. Afterwards, the number of recorded reshuffles is examined. Lastly, the number of containers that reside in the terminal is discussed.

Number of Locations The client supplies a list of locations when the system of Cofano is implemented at the client. These locations, in turn, can be used to assign containers. A common practice among clients is to report more locations than they in reality have, in order to cope with growth. One can translate the locations, as discussed, to stacks and check whether ever a container was assigned to it. Table 4.1 gives an overview of the available stacks versus the stacks used. As can be observed, no clients use all reported stacks. Disregarding client E and F, halve to a quarter of the stacks available are used. Therefore, the number of used stacks is assumed to be the actual number of stacks present at a client.

Clients E and F are different from the others, as they share the same terminal. Therefore the reported locations are the same (apart from some minor copying errors). When comparing their used stacks, 77 of the stacks are used by both terminals.

	Available	**Used**
Client B	608	325
Client C	1312	730
Client E	1085	193
Client F	1002	173
Client G	336	234

Table 4.1: Overview of the number of reported stacks versus the number of stacks used.

One can divide the number of locations by the number of containers handled per week. This figure gives insights into the density of the terminal. A high number indicates a few containers per location, whereas a low number indicates a lot of containers. This figure varies greatly between clients. Client F has the highest of 2.92 locations per container. The lowest is at Client G that has 0.234 locations per container. On overage, a client has 1 location per container.

Stack Homogeneity [Figures A.10] To get a sense of the homogeneity of a stack, one can take a specific moment in time and check whether all containers in a stack match a given feature. This measure only makes sense for stacks that contain at least two containers. Therefore the analysis is limited to the stacks of at least two high. The score is determined by taking the

[1] Translated from the original language.

stack contents of the end of the week and counting the number of non-homogeneous stacks and divide them by the total number of stacks. The following five features are analyzed:

- *Owner*: If all containers have the same owner (carrier) according to the booking.
- *Empty*: If all containers are either full or empty.
- *TEU*: If all containers are of the same TEU.
- *Reefer*: If all containers are either reefers or non-reefers.
- *Filled*: If all containers are either filled reefers or non-filled reefers (empty reefers and other containers).

Of the selected features, the owner is mixed the most, ranging just above 20%. This makes sense as there are more distinct owners than, for instance, TEU sizes, making it harder to create a stack where these are not mixed. For empty containers, the percentage usually stays below 20% at most of the clients. The distribution of filled and empty containers are the same across all clients. If containers were to be stacked randomly, one would expect at least 50% of mixed stacks. Therefore this suggests that filled and empty containers are usually kept in separate stacks. For TEU this argument is more difficult as most clients commonly only handle a single TEU size. Two clients (C and E) handle two TEU sizes with a ratio around 1:2. If stacked randomly, one would expect at least a percentage of 44%, while in reality, this lays at most around 20%. Often weeks do not even have, or close to none, mixed TEU stacks suggesting that TEU sizes are rarely mixed.

Of the analyzed clients, only two handle reefers regularly (B and F). Of these clients, Client F almost exclusively handles reefers while Client B handles both. As the number of clients that handle both reefers and normal containers is low, not much can be said about the mixing of reefers and non-reefers. However, when looking at Client B, it can be noted that reefers and non-reefers do get mixed, only if the reefer is empty.

Number of Reshuffles [Figure A.11] A reshuffle is only counted when the client reports a move from one location to another location. It might be that this move is not a reshuffle move (a move that removes a blocking container) but rather a move to prepare a container for departure. One client may retrieve a container directly from the terminal, whereas another client first moves it to a designated area from which another machine loads in onto a carrier. As a direct retrieval does not count as a reshuffle, this might be a reason why client B has a large number of reshuffles compared to other clients.

Furthermore, the number of reshuffles varies largely between weeks. Some weeks there are none, or close to zero, reported reshuffles while other weeks reach up to around 100. It might be that clients do not register all reshuffles. On the other hand, looking at the number of places available to the clients and the number of containers handled, it is not that farfetched that reshuffles do not occur that often.

Residing Containers [Figure A.12] As compared to the container relocation problem (CRP), a terminal in real-life rarely is fully empty. This means that a given point in time, containers are present at the terminal. Three measures are taken to get insights into the number of containers that remain in the terminal at a given time during the week. The first measure is the minimum number of containers that remained during the week. The second measure is the number of containers that remained at the end of the week. The third and last measure is the maximum

number of containers that remained in the terminal during the week. At a given point in time, the number of containers that remain in the terminal varies between 50 to 450 depending on the client.

The difference between min and max within a week is not that big compared to the number of containers arriving and departing. This number varies between 50 to 100 depending on the client. This indicates that the handling of in- and out-bound containers do not occur linearly, but rather in an alternating fashion.

Conclusions Between a quarter to a half of the available locations are used in stacking. Most likely, the non-used locations are reserved for growth and can not be used at the moment. Although the evidence is not stunning, it seems to suggest that empty containers are stacked together, containers of different TEUs are rarely mixed and filled reefers are rarely mixed with empty reefers. The number of reshuffles differs from week to week. Some weeks the number of reshuffles reach up to 100 while during other week none are reported. Looking at the number of available locations, it should be possible that during some weeks no reshuffles occur. Lastly, depending on the client, the number of containers that remain in the terminal at a given point in time varies from 50 till 450.

4.4 Conclusions

This chapter sought to answer the question: *"What are arrival, departure and stacking characteristics of shipping containers?"*. This was accomplished by analyzing data of seven clients of Cofano. Three aspects were analyzed: the number of handled containers, the characteristics of arrival and departure times, and the currently used stacking approaches. It was found that the clients vary greatly in the number of containers handled. Three clients handle around 1000 containers a week, while the others less than 500. All containers are handled from Monday to Friday. The average dwell time of containers is around 3 days averaged over all clients. The number of locations a client has available per weekly arriving container also varies greatly. The highest is 2.92 locations per container and the lowest is 0.234. On average this is around one location per weekly arriving container.

The conclusions found in this analysis are used to defend the assumptions made in this thesis. Furthermore, the findings are used to generate real-life problem instances to see how well the found solution performs. The next chapter will give a model definition.

Chapter 5

Model Description

This chapter will answer the second research question: *"How to model the stacking problem?"*. The goal of this question is to give a formal definition of the problem solved in this thesis. The problem at hand is a stochastic sequential decision problem. Therefore it lends itself to be modeled using SDP. As SDP is not taught in the Computer Science curriculum, first a brief introduction into SDP is given. This introduction starts with explaining DP and builds up to SDP. Readers familiar with SDP may skip Section 5.1. Afterward, eight assumptions made in this research are presented, referred to as A_1 to A_8. Based on these assumptions, a model is defined. After the model description, the used restrictions are elaborated.

5.1 Dynamic Programming

Dynamic Programming (DP) breaks the problem at hand up into smaller problems. Such problems have a set of possible states, which usually is denoted by $\mathcal{S}$. One way of breaking up the problem is by dividing it into time periods or stages. At each time step (or stage) t, the system is in a certain state $S_t \in \mathcal{S}$. In state S_t one can choose from a set of decisions, denoted by $\mathcal{A}_t$. The decision $a_t \in \mathcal{A}_t$ transitions S_t to a new state S_{t+1}. For this, the transition function $S^M(S_t, a_t)$ is defined. A decision a_t has a certain cost (or reward) given by a function, typically denoted by $C_t(S_t, a_t)$. The purpose of DP is to unravel the value of each state (given by value function $V_t(S_t)$), from which a policy can be formed.

One can describe the value of a state as the sum of the direct experienced costs plus the costs experienced later. Usually also a discount factor γ is included. This factor enables one to weigh future costs less and may be useful when, for instance, the future is uncertain. In a deterministic setting, one can write the value function as follows:

$$V_t(S_t) = \min_{a_t \in \mathcal{A}_t} \left\{ C_t(S_t, a_t) + \gamma V_{t+1}(S^M(S_t, a_t)) \right\} \tag{5.1}$$

5.1.1 Stochastic Dynamic Programming

Stochastic Dynamic Programming (SDP) is the variant of Dynamic Programming that includes uncertainty. Future costs (or rewards) may not be known when making a decision. An example

of such a problem is determining the stock prices. These prices fluctuate over time due to exogenous influences. At the moment of buying a stock, one does not know the value of that stock in the future. In addition, the direct costs may be unknown as well. An example of such a problem is route planning where congestion on the roads may suddenly arise. When deciding the road to drive on, it is unknown if congestion emerges. Thus the exact costs of the road are unknown as well. In this thesis, the problem at hand has deterministic costs (i.e. a reshuffle is either executed or not). Therefore, in the remainder of this thesis, it is assumed that direct costs are known.

A SDP problem consists, at minimum, out of the following five elements: *state variable, decision variable, exogenous information, transition function* and *objective function* [25]. The state variable captures all the information needed to make a decision as well as the information on how the system evolves. The decision variable represents the actions that can be taken. The exogenous information is the data that becomes known each period. An example of exogenous information could be the next container that is to be retrieved. The transition function describes how the system evolves given an action. The objective function specifies the costs being minimized or the rewards being maximized.

The equation from Section 5.1 (Equation 5.1) is also known as Bellman's equation and can be used to determine the optimal decision in a deterministic problem. This equation can be updated to deal with stochastic problems. Assuming the direct costs are deterministic, one can write the following formula:

$$V_t(S_t) = \min_{a_i \in \mathcal{A}_t} \left\{ C_t(S_t, a_t) + \gamma \mathbb{E} \left\{ V_{t+1}(S^M(S_t, a_t, W_{t+1})) | S_t \right\} \right\} \tag{5.2}$$

Here W_{t+1} denotes the exogenous information known between t and $t+1$. In the stochastic setting the new state is not solely dependent on S_t and a_t but also includes W_{t+1}. W_{t+1} consists out of variables (for instance buy and sell prices for stocks) that are unknown at the time of planning.

A distinction can be made between two types of states: the state when one receives information (the *post-decision* state) and the state when one needs to make a decision (the *pre-decision* state). Figure 5.1 depicts the pre- and post-decision using a decision tree. In this figure, a straight line represents an action and a dotted line the revealment of exogenous information. Using this separation of states, the transition function can be split into two functions. The first function describes the state transition when an action is taken and the other the state transition when information is received. The following notation is used: S_t denotes the state before one makes a decision and S_t^a denotes the state immediately after one makes a decision. This yields the following equations:

$$S_t^a = S^{M,a}(S_t, a_t)$$
$$S_{t+1} = S^{M,W}(S_t^a, W_{t+1})$$

Defining states and the transition functions this way, besides perhaps elegance, does not yield any direct benefits. The benefit lays that the value function (Equation 5.2) can be separated into two parts. This yields the following equations:

$$V_t(S_t) = \min_{a_i \in \mathcal{A}_t} \left\{ C_t(S_t, a_t) + \gamma V_t^a(S_t^a) \right\} \tag{5.3}$$
$$V_t^a(S_t^a) = \mathbb{E}\left\{ V_{t+1}(S_{t+1}) | S_t^a \right\} \tag{5.4}$$

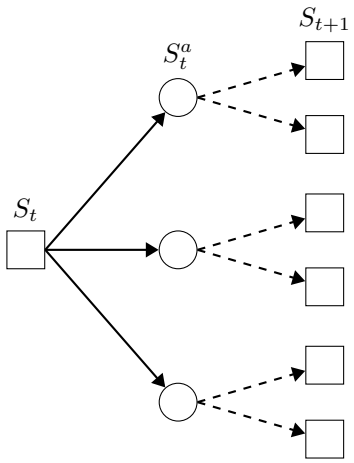

Figure 5.1: The pre- and post-decision state variables depicted as a decision tree. A square node represents pre-decision nodes, circle nodes represent post-decision nodes, a straight line depicts an action and a dashed line represents the retrieval of exogenous information.

Here Equation 5.3 is a deterministic optimization problem once the value of V_t^a is given or can be approximated. Equation 5.4 expresses the value of future states. Expressing value functions this way avoids the need for computing the expectation explicitly within the optimization problem. Besides, one might not even be able to calculate the expectation exactly, due to an unknown underlying probability distribution. In the remainder of this chapter, the above-explained theory and notation is used to model the problem.

5.2 Assumptions

In order to formally define the model used in this thesis, first, the assumptions made are explained and defended. In the next section, these assumptions are used to define the model. This thesis tackles the stacking problem where containers both arrive and depart in a single terminal. Eight assumptions are made, referred to as A_1 to A_8:

A_1 **The terminal consists of a pre-determined fixed number of stacks and tiers.** This research focuses on the operational side of the stacking problem. Therefore it assumes the layout of the terminal is pre-defined. Furthermore, due to equipment limitations and safety concerns stacks can not exceed a given height. Based on findings in Chapter 4 the max height of a stack is set to 4.

A_2 **Containers are handled one-by-one.** In practice, it might happen that a vessel is being served while a truck arrives. If sufficient equipment and personnel are available, these can be served simultaneously. As the focus is on mid-size terminals, this capacity may not be available. Furthermore, making this assumption relaxes some complexity in the problem.

A_3 **Containers arrive and depart in batches.** This research tackles the combined problem (both handling of in- and out-bound containers). Containers are grouped into batches. The order of containers within a batch is unknown. A batch is created by grouping containers that are expected to arrive or depart roughly around the same time. An example of such are containers that arrive via a vessel. It is known that they arrive at the same time, but

the exact order is not known until the stowage planning is received. The same argument can be used for train transport. Lastly, it can also be applied to truck transport when trucks are assigned to time windows. This is the case when terminals make use of Truck Appointment System.

A_4 **The order among all batches are known, within a batch not.** The order of container arrival and departures are uncertain [11]. Usually some information is known about the travel path of a container and some can be said about the relative order. To limit the complexity of the problem it is assumed that the relative order between batches is fully known. Although this may not always be true in practice, it is a step forward compared to other models that assume that the full order of containers is known [19], no order at all is known [17] or only the outbound case is tackled [10].

A_5 **In- and out-bound containers are not mixed within a batch.** Containers within a batch are not mixed and solely consist out of either in- or out-bound containers. The motivation behind this assumption is to prevent the case where a container arrives and departs in the same batch. In that case, there exists a dependency between containers in the same batch. This breaks the assumption that the order within a batch may be arbitrary, as a container can not depart before it has arrived.

A_6 **A container may only be reshuffled when it is blocking the current target container.** This is a commonly made assumption in literature [19] and is often referred to as the *restricted CRP*. It decreases the dimensionality of the problem while not losing much optimality [24].

A_7 **The costs of reshuffling is the same for every reshuffle.** The goal of this research is to minimize the number of reshuffles. Therefore only the reshuffles are counted and, for instance, the equipment costs of the reshuffle are ignored. This assumption allows stacks to be interchangeable, reducing the state space immensely.

A_8 **All in- and out-bound containers are of the same size and type.** Containers vary in size and type. From the analysis in Chapter 4, it seems that different sizes and types of containers usually are not mixed. Therefore one can split the problem of solving the problem for all types into solving it for dedicated types of containers. This, however, introduces other difficulties, such as how to split. It is decided to ignore these and limit the scope of this research to a single type of container.

5.3 Model Description

This section elaborates on the model assumed in this thesis. First the problem is explained after which the individual parts of the SDP model are presented.

Problem Instance Assumption A_3 and A_4 state that the batches are known when planning. Therefore a problem instance is specified by the batches and the containers contained within these batches. Let B be a list of all batches, such that B_i, $i \in \{1, \ldots, |B|\}$ represents the i^{th} batch. A batch is either in- or out-bound. The number of containers within a batch may be any natural number including zero. Batches are known beforehand. Therefore this problem is a finite horizon problem. The goal of this problem is to handle all containers in a given terminal layout using as few reshuffles as possible.

A terminal layout, analogous to real-life, consists of multiple bays that contain one or several stacks. A visual example of multiple bays can be found in Figure 2.3. In real-life, a bay is usually reachable from both sides. It, however, may happen that a bay is located in front of a wall (or some other immovable structure). When this is the case, the bay is only accessible from one side. Following the classification scheme of Lehnfeld and Knust [19] the problem is classified as: $LU, b = 4|(\mathcal{I}^{in}, \mathcal{I}^{out})_K, S_{i,j}|\mathbb{E}(\#RS)$. Here $S_{i,j}$ is defined in Section 5.4.

State A state consists out of the containers that reside at the terminal and their assigned locations. At each stage $t \in \{1, \ldots, |B|\}$ a single batch is handled at the terminal. Analogous t dictates the current batch to be handled. Let L describe the current residing containers in state S_t. Let L_b, $b \in \{1, \ldots, \#bays\}$ denote the containers in an individual bay and let $L_{b,i}$, $i \in \{1, \ldots, \#stacks_b\}$ denote the containers of the i^{th} stack in bay b. A state S_t is defined as a the tuple $< L, \hat{B}_t >$, where $\hat{B}_t$ denotes the realized order of batch B_t.

Let $c_{b,o}$ denote an individual container. A container is always an element of a single inbound batch. When this batch is handled, the container arrives at the terminal. A container always has a batch number b. This batch number refers to the batch in which the container leaves the terminal. The batch number may be bigger than the total number of batches ($b > |B|$). In this case the container does not leave the terminal within the current planning horizon. The order number o is not known for the majority of the time. This number indicates the relative order within a batch and is only revealed just before handling batch b. Furthermore, each container has a unique identifier to distinguish between containers with the same batch numbers.

Decision In the defined model, three types of container moves are identified: the in-bound move (i.e. placing a container in the terminal), the out-bound move (i.e. retrieving a container from the terminal) and a reshuffle move (i.e. move a container within the terminal). Let $\tau(L_{b,i})$ be the topmost container of $L_{b,i}$ and $L_{b,i} \setminus \tau(L_{b,i})$ denote the extraction of the topmost container in $L_{b,i}$. Let $L_{b,i} + c$ denote placing a container c in $L_{b,i}$. Then the three type of container moves can be defined as:

$$
\begin{aligned}
inbound\_move(L, (b,i), c) &= \{L_{b',i'} \mid (b',i') \neq (b,i)\} \cup \{L_{b,i} + c\} \\
outbound\_move(L, (b,i)) &= \{L_{b',i'} \mid (b',i') \neq (b,i)\} \cup \{L_{b,i} \setminus \tau(L_{b,i})\} \\
reshuffle\_move(L, (b,i), (b',i')) &= \{L_{c,j} \mid (c,j) \neq (b,i), (c,j) \neq (b',i')\} \\
&\quad \cup \{L_{b,i} \setminus \tau(L_{b,i})\} \\
&\quad \cup \{L_{b',i'} + \tau(L_{b,i})\}
\end{aligned}
$$

All containers within a batch B_t are handled during stage t. Let a_t denote a decision at time t. Handling a single container may require multiple actions when a reshuffle is required. Therefore a_t denotes a sequence of container move sequences. Let $a_{t,k}$ denote the sequence of container moves for handling the k^{th} container in batch t. Due to assumption A_6, if B_t is inbound or if no containers are blocking container $B_{t,k}$, $|a_{t,k}| = 1$. If there are blocking containers, then $|a_{t,k}| = \#nr\_blocking + 1$. Note that the definition of a blocking container varies between terminal layouts. When solely using a gantry crane, containers can only be blocking when located on the same stack. When using a reach-stacker, containers do not have to be located in the same stack to be blocking. The notion of blocking containers is more thoroughly elaborated in Section 5.4.

Information and Transition Function Two types of states can be identified in a single stage. The first type of state is the one where exogenous information needs to be retrieved (i.e. the order of a batch needs to be revealed). The second type of state is where a decision needs to be made (i.e. the sequence of sequences to handle all containers within a batch). As discussed in Section 5.1, these states are referred to as, respectively, the pre- and post-decision states and allow one to split the transition function. In the remainder of this thesis, for the sake of clarity, the pre-decision state is referred to as S_t. The post-decision state is referred to as S_t^a.

Let W_{t+1} denote the exogenous information made available after handling B_t. In the proposed model, this information solely consists of the order of the next batch to handle. Let $\zeta_{t,k}$ be a random variable taking values in $\{1, \ldots, |B_t|\}$, such that $\{\zeta_{t,k} = i\}$ is the event that $c_{t,i}$ is the k^{th} container to be in handled in B_t. For now it is assumed that all containers have the same probability of being the k^{th} container within a batch. The distribution of $\zeta_{t,k}$ is given by

$$\mathbb{P}[\zeta_{t,k} = i] = \begin{cases} \frac{1}{|B_t|}, & \text{if } 1 \leq i \leq |B_t| \\ 0, & \text{otherwise} \end{cases}$$

Let S^M denote the transition function. As discussed in Section 5.1, the transition function is split into two parts. Let $S^{M,a}$ be the transition function that transforms a pre-decision state into a post-decision one (i.e. handling a batch) and let $S^{M,W}$ be the transition function that transforms a post-decision state into a pre-decision (i.e. the disclosure of exogenous information). Then the transition functions are defined as:

$$S_t^a = S^{M,a}(S_{t,1}, a_t)$$
$$S_{t+1,1} = S^{M,W}(S_t^a, W_{t+1})$$

Using the transition function, the value functions can be defined as:

$$V_t^a(S_t^a) = \mathbb{E}_{\zeta_{t,1},\ldots,\zeta_{t,|B_t|}}\left[V_{t,1}(S^{M,W}(S_t^a, W_{t+1}))\right]$$
$$V_{t,1}(S_{t,1}) = \min_{a_t \in \mathcal{A}_t}\left\{C(S_{t,1}, a_t) + V_{t+1}^a(S^{M,a}(S_{t,1}, a_t))\right\}$$

Here $C(S_{t,1}, a_t)$ is defined as the total number of reshuffles taking places while executing action a_t. Formally, it can be defined as:

$$C(S_{t,1}, a_t) = \sum_{k=1}^{|a_t|} |a_{t,k}| - 1$$

5.4 Stacking Restrictions

The previous section described the SDP model. That section mentions the decision space $\mathcal{A}_t$, but no clear definition on the restrictions is given. Some decisions are restricted by the nature of the problem. An example is that stacks, due to safety concerns and machine limitations, can not be stacked infinitely high. Also, a container can not be placed mid-air. Furthermore, the use of a reach-stacker imposes some additional restrictions as it is not able to retrieve all top-most containers, whereas a gantry crane can. This section will define the restrictions imposed by a reach-stacker as well as introducing an artificial restriction that limits the size of the decision space.

5.4.1 Reach-Stacker Restrictions

A reach-stacker is not able to retrieve every top-most container and is dependent on the height of neighbouring stacks. Besides, different brands and types of reach-stackers may influence which container may or may not be retrieved. In this thesis, it is assumed that only stacks contained within the same bay influence the reachable stacks of that bay. Furthermore, if a bay is only reachable from one side, it is assumed that is reachable from the left. Figure 5.2 shows part of a bay viewed from the side in 2D. The target container/location is indicated using the letter T. The arrows indicate where containers are allowed to be stored for T to still be reachable. If the bay is only accessible from the left, the reach-stacker can reach T when there are no containers above the solid line. I.e. the reach-stacker can reach T when there are only containers on or below the solid line (ignoring stacks behind T). All containers above this line are blocking and need to be removed before T can be accessed. This changes when the bay is accessible from both sides. If the bay is accessible from the other side, containers are allowed to be above the solid line, as long as they remain below the dotted line.

Furthermore, when reshuffling, an additional constraint is introduced. Assume container T, in Figure 5.2, is the current target container. To prevent the possibility to get stuck in an infinite reshuffle loop, it is not allowed to cause a new direct reshuffle. Therefore, it is not allowed to reshuffle a container above the defined diagonal arrows, as doing so, may introduce a new reshuffle.

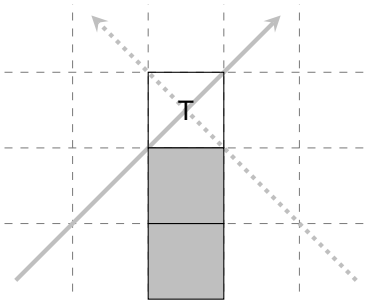

Figure 5.2: Figure showing part of a bay viewed from the side in 2D. The gray squares represent a container. The white square with the letter T represents a container or an empty spot. The arrows indicate where containers are allowed to be placed for T to be reachable.

5.4.2 Artificial Restriction

Even for small problem instances, the size of the decision space is too big to calculate in reasonable time. To limit the size, an additional stacking restriction is included. This stacking restriction is based on real-life practices and is best explained with an example. Imagine a bay that is reachable from both ends with a container placed on both outer ends. Although the stacks in between these containers are empty, they can not be used as the reach-stacker can not reach these locations. Placing containers in such a way is inefficient in terms of space usage and consequently in practice rarely done. Therefore, an artificial restriction is added that prevents such container placements. The rule makes a distinction between bays that are reachable from one way and from both ways.

For the one-way bay, first, the check is done whether the bay is empty. If the bay is empty, the container must be placed on the rightmost stack. If the bay is not empty and the container is

placed on a currently empty stack, it is not allowed to have empty stacks to the right. If the target stack is non-empty, it is a valid store location according to this restriction.

For the two-way bay, again first is checked whether the bay is empty. If the bay is empty, the container must be placed in the middle. If the bay is not empty and the target stack is empty either one of the two following statements must hold:

1. The stack to the right has a container and all stacks to the left are empty.
2. The stack to the left has a container and all stacks to the right are empty.

Similar to the one-way bay, if the target stack is non-empty, it is a valid store location according to this restriction.

With the above-explained restrictions, a terminal operated by reach-stackers can be described. Using a small trick, also a terminal that operates with gantry cranes can be described. This trick makes use of the fact that a reach-stacker can always access the top-most container/location if there is only a single stack within that bay. Therefore a terminal using gantry cranes can be modeled as a terminal using reach-stackers where each bay has only a single stack.

5.5 Conclusions

This chapter sought to answer the question: *"How to model the stacking problem"*. This was accomplished by first introducing the reader into the domain of SDP. Afterward, eight assumptions made in this thesis were elaborated, after which the model was given. Lastly, the restrictions caused by using a reach-stacker were defined as well as one additional restriction to reduce the decision space. Also, a method was introduced to model a terminal using gantry cranes as a terminal that operates using reach-stackers.

The model defined in this chapter will be used in the remainder of this thesis. The next chapter will describe approaches on how to solve the model and compare these approaches.

Chapter 6

Proposed Solution

This chapter will answer the fourth research question: *"How to apply ADP to the described model?"*. Similar to the previous chapter, first, an introduction to some basic concepts concerning ADP are given. This introduction includes the basic recipe for any ADP approach. Readers familiar with these concepts can skip Section 6.1. After the introduction, this chapter continues by explaining an optimal policy used in this thesis to determine the optimal solution of a problem instance. Optimal in this context refers to the policy that minimizes the expected number of reshuffles. Furthermore, different variations of ADP are explained and applied to problem instances to evaluate the efficiency of the approach. All data and code used can be found at: https://github.com/boschma2702/ContainerStacking.

6.1 Policies and ADP Basics

This section first introduces the notion of a policy and explains different categories of policies for solving a dynamic programming model. Afterwards, the basics concepts of ADP are explained.

6.1.1 Policy

A policy is a rule (or function) that determines a decision given the available information in a specific state S_t [25]. This decision can be based on simple rules (i.e. if this, then pick that action) or based on the computed value of the current state and ones to follow. One can write the optimal decision at time t as:

$$a_t = \arg\min_{a \in \mathcal{A}_t} \left\{ C_t(S_t, a_t) + \gamma \mathbb{E}\left\{ V_{t+1}(S^M(S_t, a_t, W_{t+1})) | S_t \right\} \right\} \tag{6.1}$$

It can be noted that Equation 6.1 is simply Equation 5.1 only now one is interested in the decision that leads to the minimal value of a state. Let Π denote the set of all policies and $A^\pi(S_t)$ the function that gives the action according to policy π for state S_t. Then the optimal (or exact) policy

over a planning horizon T (which could be infinite) can be written as:

$$A^{optimal}(S_{t'}) = \arg\min_{\pi \in \Pi} \sum_{t=t'}^{T} \gamma^{t-t'} C_t(S_t, A_t^{\pi}(S_t)) \tag{6.2}$$

Besides the optimal policy, various other policies can be defined. Powell [25] identifies four broad categories of policies: *myopic policies*, *lookahead policies*, *policy function approximations* and *value function approximations*. These categories are explained in the next sections to give a sense of the possibilities for solving dynamic problems.

Myopic Policies

One of the most straightforward policies, which bases its decision on a cost function, is the myopic policy. The myopic policy only takes direct costs into account and works well for problems where future costs are less important than direct costs. A decision, following the myopic policy, is chosen using:

$$a^{myopic}(S_t) = \arg\min_{a \in \mathcal{A}_t} C(S_t, a) \tag{6.3}$$

Lookahead Policies

More sophisticated policies than the myopic ones are the lookahead policies. The lookahead policies optimize over some time horizon. This way future costs are included in a decision. A downside of the lookahead policies is that they are computationally expensive. Three widely used lookahead policies in engineering practice are tree search, roll-out heuristics and rolling horizon procedures [25].

Tree Search Tree search enumerates all possible actions and all possible outcomes over some time horizon. Because some problems explode even over a small horizon, this approach is limited to specific problems or very small horizons. If a full tree search over a certain time horizon can be performed, an optimal decision will be found.

Roll-out Heuristic For some problems, tree search might not be applicable. A way to cope with this problem is to use a heuristic policy to evaluate a single path from a given state to get an idea of the value of that state. Decisions during evaluation are made based on some heuristic. The costs experienced dictate the value of a path. The first decision made of the most promising path is used to make the actual decision. The advantage of this approach is that it is simple and gives better approximations than only using the heuristic policy.

Rolling Horizon Procedures One way of dealing with the exploding state space is to reduce the time horizon over which one tries to calculate a solution. Assume the problem is solvable over a small horizon H. Then the problem is solved by calculating the problem from t to $t + H$. This calculation yields a decision a_t that is used to calculate the solution over $t+1$ till $t+H+1$. A problem arising from this approach is that the horizon might be limited to small horizon H.

Policy Function Approximations

Policy function approximations return directly a decision given a state without solving an underlying optimization problem. They take a specific aspect of the problem and base their decision around it. These kinds of policies are useful when there is a clear idea of how to solve a problem. Take, for instance, the following problem: A police officer needs to write tickets for speeding cars. The goal is to optimize the income from the fines. Writing a ticket takes a certain amount of time and the fine is proportional to the speeding. The officer can only write one ticket at a time. More people speed by a little (low fine) than people who speed by a lot (high fine). An example of a policy function approximate is defining the policy such that the officer only writes a ticket when the speeding is above a certain threshold of s. Instead of formulating the problem as a dynamic one and solving it, it now becomes one of finding the optimal value for s.

Value Function Approximations

Value function approximations rely on approximations of the value of future states. Rather than calculating the value of each state exactly, an approximation is used. Solving an optimization problem remains the same as solving a dynamic program. The difference is that the downstream costs are now estimated by a value function approximation rather than calculated exactly.

6.1.2 Approximate Dynamic Programming

Approximate Dynamic Programming (ADP) can be used to solve various large scale discrete-time multistage stochastic control processes [21]. It can be seen as a framework for dealing with large problems where one can use various approaches and techniques. Powell [25] describes three “curses of dimensionality”. These “curses” make it difficult for some problems to calculate an exact solution. The first curse is that the state space $\mathcal{S}$ is too large to evaluate the value functions $V_t(S_t)$ for each state in a reasonable time. The second curse is that the decision space $\mathcal{A}_t$ may be too large to find a solution in a reasonable time. Third and lastly, computing the expected future costs may be intractable when the outcome space is too large.

Equation 5.2 gives a way of solving problems exactly. For a lot of problems, this is not feasible due to one (or more) of the “curses of dimensionality”. To overcome these curses, one can try to approximate rather than trying to solve the problem exactly. ADP does this by running a simulation that steps forward in time. To be able to step forward in time, two problems need to be solved: How to make a decision and how to create samples of what the exogenous information might become.

To make a decision one can take Equation 6.1. This equation gives already a way to determine an action. The only problem is that one does not know the value of $V_{t+1}(S^M(S_t, a_t, W_{t+1}))$ and probably is unable to calculate it (due to dimensionality of the problem). To overcome this problem, an approximate value function, denoted by $\bar{V}_{t+1}(S_t)$, can be used. This yields the following formula:

$$a_t = \arg\min_{a \in \mathcal{A}_t} \left\{ C(S_t, a) + \gamma \mathbb{E} \left\{ \bar{V}_{t+1}(S^M(S_t, a, W_{t+1})) | S_t \right\} \right\} \tag{6.4}$$

What remains to solve is that one does not know the exogenous information that enters the system at time t. This problem can be solved by introducing sample paths. These samples can be, for instance, obtained by using historical data or be sampled from a known distribution. A sample is needed for every t in the planning horizon. Let ω be a path of samples and $W_t(\omega)$ the

sample at time t. As a single sample path does not yield much information, multiple samples are generated. In the remainder of this thesis, a superscript n is used to distinguish between different iterations. $W_t(\omega^n)$ thus indicates the sample at time t in iteration n. The next step in ADP is to evaluate the generated sample paths. During an evaluation, the value of a state is observed (denoted by $\hat{v}_t^n$). This observation is used to update the estimated value of that state. Let $\bar{V}_t^n(S_t)$ denote the estimate of the value for state S_t at iteration n (where $\bar{V}_t^0(S_t)$ is the initial approximation). Algorithm 1 outlines the basis of the ADP algorithm. This version of the ADP algorithm is commonly referred to as the single pass approach.

Algorithm 1 Approximate Dynamic Programming Single Pass algorithm

Choose initial state S_0^1
Choose initial state $\bar{V}^0$
$n = 1$
while $n < N$ **do**
  Choose sample path ω^n
  for $t = 0, \ldots, T$ **do**
    Solve

$$\hat{v}_t^n = \min_{a \in \mathcal{A}_t} \left\{ C(S_t, a) + \gamma \mathbb{E} \left\{ \bar{V}_{t+1}^{n-1}(S^M(S_t, a, W_{t+1}(\omega^n))) | S_t \right\} \right.$$

    and let a_t^n be the value of a_t which solves the statement
    update $\bar{V}_t^n$ using $\hat{v}_t^n$ and $\bar{V}_t^{n-1}$
    compute $S_{t+1}^n = S^M(S_t^n, a_t^n, W_{t+1}(\omega^n))$
  end for
  For all non-visited states, update $\bar{V}_t^n$ using $\bar{V}_t^{n-1}$
  n = n + 1
end while

The core of this algorithm is solving $\hat{v}_t^n$. The calculated value, in turn, is used to update our approximation of the current state. The action which solved the equation for $\hat{v}_t^n$ is used to determine the next state to explore. Various problems and challenges remain for this algorithm. The first is that N needs to be big enough to get good approximations. The second problem is that the values of states are based on estimates of future states. A downside of this is that costs experienced downstream may take many iterations before they are reflected into earlier time periods. To overcome this problem, a double pass approach can be used [25]. The outline of the double pass algorithm is the same as the Algorithm 1, but the update of estimates is done after an entire sample path is explored. The observed values are determined in reverse order of visiting using $\hat{v}_t^n = C(S_t^n, a_t^n) + \lambda \hat{v}_{t+1}^n$, with $\hat{v}_{T+1}^n = 0$.

A third problem in this algorithm is to get the expected value of the approximation function. The possible outcomes may be too big to calculate this value. This can be solved by running Monte Carlo simulations to get an idea of the expected value. A different approach makes use of the post-decision state. Rather than calculating the expected value of the pre-decision state approximations, one can include the expected value into the approximation function by making an approximation of the post-decision state. Figure 6.1 shows the difference between both ways of approximating using a decision tree. To apply post-decision approximation, the calculation of

$\hat{v}_t^n$ is replaced by the following equation:

$$\hat{v}_t^n = \min_{a \in \mathcal{A}_t} \left\{ C(S_t, a) + \gamma \bar{V}_t^{a,n-1}(S^{M,a}(S_t, a)) \right\} \tag{6.5}$$

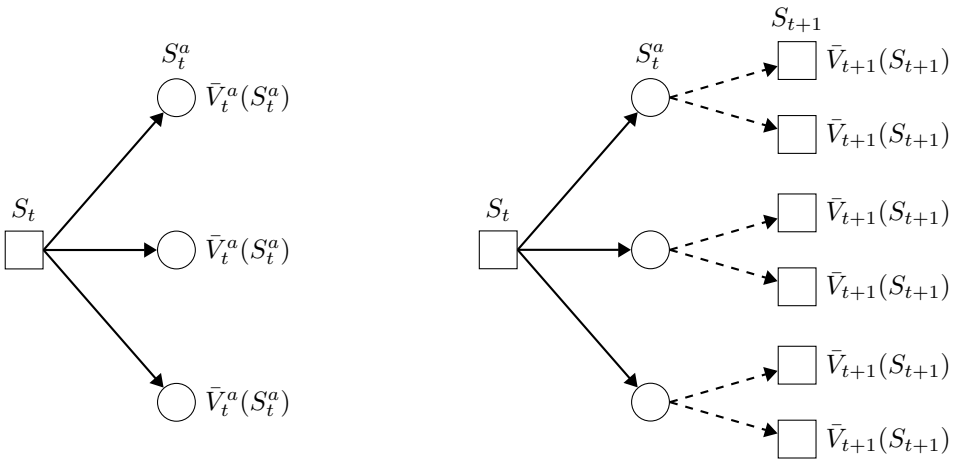

Figure 6.1: Figures showing the difference between approximate function on the pre- and post-state variable. The left image shows the approximate value function applied to the post-decision state. The right image shows the approximate value function applied to the pre-decision state variable.

A fourth problem is that currently, the algorithm only visits states of which it thinks are the best. This is referred to as an exploiting algorithm. It might be that the algorithm is stuck in a local optimum, and thus has not learned the optimal solution. An example of when this occurs is when the initial value estimate of states is an overestimate. This way, only explored paths will be visited again, as the observed values are lower than the initial overestimate. On the other hand, when picking an underestimate, the algorithm will follow a pure exploring behaviour until it has visited all states. It is unlikely that the algorithm will explore all states and thus this approach is unusable in practice. To strike a balance between exploring and exploiting, one can let the algorithm choose a random decision with a certain percentage. This method is known as ϵ-greedy. The value of ϵ indicates the percentage of random choices.

A fifth problem is how to update the value of $\bar{V}_t^n$ using $\bar{V}_t^{n-1}$ and $\hat{v}_t^n$. During learning, the algorithm observes new values. These values are used to update the old values. The old value might still be an initial guess or a value that has been updated over multiple iterations. It is desired to have a value that is based on multiple iterations. The downside of this is that in the early iterations one might experience values that are far of due to poor choice of the non-converged algorithm. Weighing the current observation equal to all previous might lay to much influence on the current observation (it might be an explorative observation). The value approximations are updated according to $\bar{V}_t^n = (1-\alpha^n) * \bar{V}_t^{n-1} + \alpha^n * \hat{v}_t^n$. Here α^n determines the weight of previous observations against the current observation. Several methods are available to change α^n over time. This thesis explores two stepsize methods: *Fixed* and *Harmonic*. The fixed stepsize reduces the alpha by a fixed amount every iteration. The harmonic stepsize is given by $\alpha^n = \max\{\frac{a}{a+n-1}, \alpha^0\}$. here a determines the rate at which the stepsize drops to zero. A larger a decreases the rate, while smaller increases it. Powell [25] suggest picking an a, such that the alpha equals to 0.05 at point of termination.

The remainder of this chapter will first describes the optimal policy used in this thesis. Afterwards, it will continue discussing various ADP solutions as the exact solution is not applicable to larger problem instances.

6.2 Optimal Algorithm

The optimal algorithm is defined as the policy that chooses the decision with the lowest expected reshuffles. In other words, a policy that follows Bellman's Equation when making a decision (Equation 5.2). Simply enumerating all possible outcomes and states would already be infeasible for tiny instances. Therefore the optimal policy proposed in this thesis follows techniques used in the PBFS algorithm of Galle et al. [10]. The PBFS algorithm is designed for the unloading problem, whereas this thesis tackles the combined problem. Therefore some adaptations are made to this algorithm while preserving its key techniques.

Four key techniques are used in the PBFS algorithm. The first is, first exploring promising nodes as they are more likely to result in the optimal solution. A promising node is defined as a node that has a small lower bound on the number of reshuffles. The second technique used is to apply A^*-search when the complete order is known (no more exogenous information is received in the system). The third technique is pruning using a lower bound function in order to detect the suboptimality of nodes before actually calculating them. The fourth and final technique used is abstraction. The first three features could be used without significant alteration. The fourth technique, however, needed rework. In the next section, the abstraction method is explained in more detail after which the algorithm itself is presented.

6.2.1 Abstraction

The state-space of the problem grows exponentially large in the number of containers and available stacks. When only interested in the number of reshuffles, some states can be seen as equivalent [18]. Take, for instance, a terminal, operated using gantry cranes, with n empty stacks. It does not matter in which stack a container is placed, as the result obtained from one location can be applied to the others. Therefore, the number of states that need to be explored, can be reduced by reusing the solution found for equivalent states. It now remains how to determine whether two states A and B are equivalent. For this, an abstraction function is introduced. The following must hold for this function:

$$abstract(A) = abstract(B) \implies A \approx B \tag{6.6}$$

Galle et al. [10] introduces an abstraction based on the sorting of individual stacks. This approach will not work in the proposed model, as stacks now belong to bays. Therefore a new abstraction is defined in this thesis. The new abstraction is based on the sortability of bays, stacks and containers. Therefore first the order among these entities is defined. A container is smaller than another container when its batch number is smaller than the other or when the batch numbers are the same and the order number is smaller. The comparison between stacks is a height-wise comparison between the containers in both stacks starting at the lowest tier. An empty spot is smaller than a non-empty one. Bays are ordered based on two features. First, it is compared whether the bay is reachable from both sides. A bay that is not reachable from both side is smaller. Then the stacks within the bay are compared pair-wise. A non-existing stack is smaller than an existing one. If a bay is reachable at both sides, it is mirrored and compared to

itself. The smaller one is chosen in the abstraction. An abstracted terminal is a terminal where all bays are sorted in ascending order. An example of the abstraction is visualized in Figure 6.2.

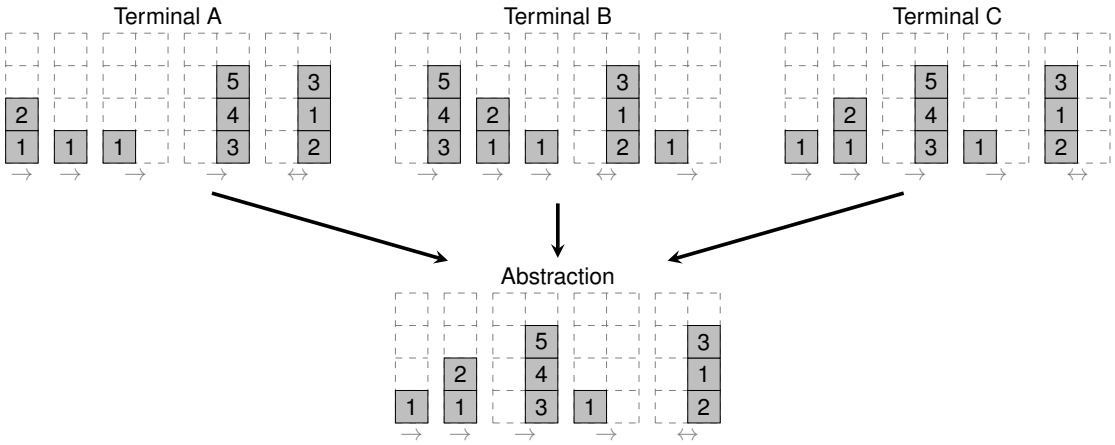

Figure 6.2: Figure showing three different terminal layouts and their corresponding abstraction. The arrow indicates from which side the bay can be accessed. Terminal A has only unordered bays within the same reachability. Terminal B has bays with different reachability unordered. Terminal C has unordered stacks within a two-way reachable bay.

6.2.2 The Algorithm

The optimal algorithm takes two arguments. The first is a node of the tree that is being explored. This node contains all information of the current terminal layout, as well as the batch information. The second argument is the lower bound function used to prune on. An outline of the algorithm is shown in 2.

Algorithm 2 Adapted PBFS Algorithm

```
1: function OPTIMAL(n, l)
2:     if n is post-decision state then
3:         for realization in all realizations do
4:             value = 0
5:             if an abstraction of n already calculated then
6:                 value += previous calculated value
7:             else
8:                 apply realization to n
9:                 value += exact(n, l)
10:            end if
11:        end for
12:        return value * |realizations|^-1
13:    end if
14:    if n is pre-decision sate then
15:        if current batch is last batch then
16:            return A*(n)
17:        end if
18:        outcomes = generate all possible outcomes handling current batch
19:        Sort outcomes using l
20:        k = 0
21:        current_best = exact(outcomes[k], l)
22:        while k < |outcomes| and l(outcomes[k]) < current_best do
23:            if an abstraction of outcomes[k] already calculated then
24:                new_value = previous calculated value
25:            else
26:                new_value = exact(outcomes[k], l)
27:            end if
28:            current_best = min(current_best, new_value)
29:            k += 1
30:        end while
31:        return reshuffles + current_best
32:    end if
33: end function
```

At the end of the algorithm, the expected number of reshuffles is returned. As a byproduct, it saves all the values of visited states. This byproduct can be used to determine the optimal stack assignments for containers given a realization. This can be accomplished by generating all possible outcomes (per batch) and selecting the outcome with the best value.

6.3 ADP Solutions

The total number of realizations possible for a single batch of size n is equal to $n!$. This means that the total number of unique realizations over the planning horizon T is equal to the product of the number of realizations of each batch ($\prod_{t=1}^{T} |B_t|!$). This number quickly becomes too big to calculate as the number of batches and/or batch sizes grow. Therefore, in this thesis, an ADP solution is proposed.

As discussed in Section 6.1.2, ADP is a flexible solution with lots of possible adaptations and tuneable parameters. As time is limited, only a few adaptations are explored and tuneable parameters are fixed. All algorithms described in this thesis are run using a $\lambda = 1$ (discount factor) and $\epsilon = 0.05$ (epsilon greedy). The remainder of this section is structured as follows. First, the method used to evaluate the algorithms is described. Afterwards, two basic ADP algorithms are explained and evaluated. Finally, a more complex ADP approach is introduced and performance examined.

6.3.1 Evaluation

All algorithms described in this thesis are executed on the same problem instances. As running lots of simulations on bigger instances takes to much time, the algorithms are tested on smaller instances. In these instances, time is abstracted to a linear integer scale. A problem instance is generated using the following approach:

1. 25 containers are given a random arrival time between 0 and 20 (uniformly distributed).
2. For each container, a random dwell time is sampled using an exponential distribution with mean 12.
3. The containers are grouped into batches using their arrival and departure time. The departure time is calculated by adding the arrival and dwell time.
4. 150 random realizations are generated for training.
5. 150 random realizations are generated for evaluation.

If one resembles a single time unit as an hour and assumes a working day consists out of 8 hours, the planning horizon covers 2.5 days with an average dwell time of 1.5 days. By generating the random realizations, each algorithm can be trained and evaluated using the exact same set of realizations. This makes the comparison between algorithms fairer. In total, 16 problem instances are generated and used for evaluation.

Each ADP algorithm runs every problem instance on two distinct terminal layouts. The first terminal layout represents a terminal where all stacks can be served using a gantry crane. In total, this terminal consists out of 7 stacks (and thus 7 bays). The second terminal layout represents a terminal where all stacking is done using reach-stackers. This terminal consists out of 3 two-way bays, each with 5 stacks. Both layouts have a maximum stack height of four. In the remainder of this thesis, the terminals will be referred to as terminal layout 1 and terminal layout 2 respectively.

Each ADP algorithm runs 150 iterations using the generated training realizations. Each 10^{th} iteration, starting after the first iteration, an evaluation is executed. Every evaluation solves all 150 evaluation samples and decisions are made exploiting what the ADP algorithm has learned so far. In total, for a single ADP and terminal combination, $16 * 150 * 15 = 36000$ evaluation simulations plus $16 * 150 = 2400$ learning simulations are run. During each evaluation, two things are recorded: the average number of reshuffles that occurred during the 150 evaluation simulations and the approximate value of the initial state (S_0). The first measure indicates the performance of the algorithm, whereas the second measure indicates the convergence of the algorithm. The averages over the 16 problem instances of these measures are reported in the thesis.

In order to get insights in how well the ADP approaches compare to the optimal solution, the

optimal algorithm described in Section 6.2 is applied to all 16 problem instances using the same terminal layouts. Terminal layout 1 could be solved using 0 reshuffles for all problem instances. The optimal algorithm did not yield an answer for terminal layout 2 as it ran out of memory during calculation.

6.3.2 Basic ADP Solutions

First, the difference of the double and single pass ADP approaches, described in Section 6.1.2, are evaluated. A lookup table is used to store the values for each state. Rather than the concrete state, the abstracted state (following the technique described in 6.2) is stored in a lookup table. Values are initialized at 0. Two different stepsizes are used. The first stepsize is the Fixed stepsize with $\alpha = 0.05$. The second stepsize is the Harmonic stepsize with $a = 1$ and $\alpha^0 = 0.05$. The single and double pass algorithm are run with both stepsizes. The results are shown in Figure 6.3. The results of terminal layout 2 could not be obtained, as the algorithm ran out of memory trying to solve the problem instances.

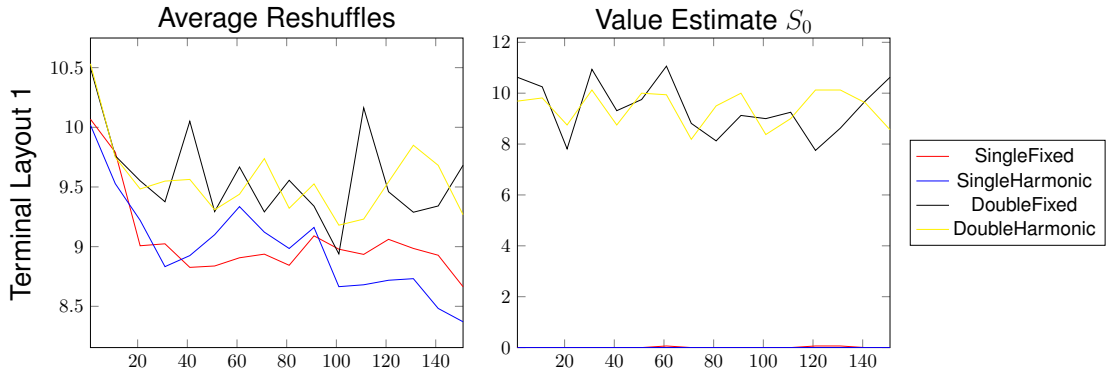

Figure 6.3: Graph showing the results of the single and double pass algorithm using the Fixed ($\alpha = 0.05$) and Harmonic ($a = 1$ and $\alpha^0 = 0.05$) stepsizes. The graph left shows the average number of reshuffles at each iteration. The graph right shows the value estimate for the initial state.

As can be observed in Figure 6.3, the single pass algorithm, on average, performs one reshuffle less than the double pass algorithm. However, both algorithms perform way off the optimal solution of zero. The value approximate of the initial state remains zero after 150 iterations for the single pass. This could be caused by two things. The first is that the costs experienced in the future are not propagated back yet to the initial state. The second cause could be that there are still unexplored states directly reachable from the initial state. These states have an initial value of zero, resulting in a value of zero for the initial state. This problem is not apparent in the double pass. Therefore, the remainder of the ADP solutions presented in this thesis make use of the double pass version of ADP. Furthermore, there seems to be no significant different between the different stepsizes.

6.3.3 Basis Function Solutions

When looking at Figure 6.3, it can be observed that the algorithm converges slowly. This is because the algorithm needs to explore all unique abstract states to come up with a good solution. This is infeasible for larger problem instances. To solve this problem, a value function that generalizes across states is needed. A way of accomplishing this is using basis functions.

The idea behind basis functions is that it extracts certain features from a state and uses the value of these features to come up with an estimate. The efficiency of basis functions relies on the identification of characteristics or features that define the true value of a state.

Let $\mathcal{F}$ denote the set of features. The value of for a state for each feature $f \in \mathcal{F}$ can be obtained by evaluating the basis function $\phi_f(S_t)$. These values, in turn, can be combined in a linear combination using a weight θ_f^n for each feature f as follows:

$$\bar{V}_t(S_t) = \sum_{f \in \mathcal{F}} \theta_f^n \phi_f(S_t) \tag{6.7}$$

The weight θ_f^n is updated in each iteration n. As the problem tackled in this thesis is a finite horizon problem, each t has its own set of weights. Several methods are available to tune the weight $\theta_{t,f}^n$ for each feature $f \in \mathcal{F}$. In this thesis, the recursive least-squares method is chosen as this is an effective approach [21]. Two flavours of this method are available, for *nonstationary* and *stationary* data. The nonstationary provides the opportunity to put more weight on more recent observations, whereas the stationary puts equal weights on all observations. Details on how least-squares updates the weights, can be found in [25]. Equations used in this research are noted down below. The weights θ_t^n are updated each iteration using:

$$\theta_t^n = \theta_t^{n-1} - H_n \phi^n(S_t) \left(\bar{V}_t^{n-1}(S_t) - \hat{v}_t^n \right)$$

where H^n is a matrix

$$H^n = \frac{1}{\gamma^n} B^{n-1}$$

and where B^{n-1} is a $|\mathcal{F}|$ by $|\mathcal{F}|$ matrix that is updated using:

$$B^n = \frac{1}{\alpha^n} \left(B^{n-1} - \frac{1}{\gamma^n} \left(B^{n-1} \phi^n(S_t) (\phi^n(S_t))^T B^{n-1} \right) \right)$$

and γ^n is

$$\gamma^n = \alpha^n + (\phi^n(S_t))^T B^{n-1} \phi^n(S_t)$$

Matrix B^n is initialized using $B^0 = \rho \mathbb{I}$, where $\mathbb{I}$ is the identity matrix and ρ is a small constant. This initialization works well when the number of observations is large [25]. In the remainder of this thesis, to limit the number of tunable parameters, the value of ρ is set to 0.1.

Analogous to the Fixed and Harmonic stepsize, the value of α^n determines the weight on the prior observations. Setting α^n to 1 for all observations yields the least-squares method for stationary data. Lower values of α^n decrease the weights on prior observations. α^n is defined as:

$$\alpha^n = \begin{cases} 1, & \text{, stationary} \\ 1 - \frac{\delta}{n}, & \text{, nonstationary} \end{cases}$$

where δ is a tunable constant variable. Setting $\delta = 0$ yields equal weights on all observations, whereas a lower value decreases the weight on later observations.

Feature Functions

The basis function relies on the features used. In total, nine features are selected and applied in this thesis. Using these features, five value function approximates (VFA's) are defined. Table 6.1 shows the combination of features used for each of the VFA and their corresponding name used in the remainder of this thesis. Each feature is discussed in more detail below.

Feature	VFA1	VFA2	VFA3	VFA4	VFA5
1. Blocking Lower Bound	X	X			X
2. Unordered Stacks	X			X	X
3. Composite Measure	X		X		
4. Batch Label Difference	X		X	X	
5. Avg Stack Height	X	X			
6. Non-Reachable Stacks	X	X	X	X	X
7. Non-Reachable Containers	X			X	
8. Future Blocking Stacks	X	X			
9. Future Blocking Containers	X		X		

Table 6.1: Table showing which VFA uses which feature functions.

Blocking Lower Bound The Blocking Lower Bound feature (BLB) is based on the lower bound described by Galle et al. [10]. It describes the number of expected reshuffles at least needed to empty a given stack. It serves as a lower bound on the total number of reshuffles. Let $H(s)$ denote the height of stack s and s_i the container at tier i (where $i = 1$ represents the lowest container), then BLB is defined as:

$$BLB(L) = \sum_{b \in L} \sum_{s \in L_b} \left\{ H(s) - \sum_{h=1}^{H(s)} \frac{\mathbb{1}\{s_h = \min_{i=1,\dots,h}(s_i)\}}{\sum_{i=1}^{h} \mathbb{1}\{s_h = s_i\}} \right\}$$

A high BLB value indicates a bad state as the number of expected reshuffles is high.

Unordered Stacks The Unordered Stacks feature (US) counts the total number of stacks that (may) cause a reshuffle. This feature is similar to BLB, only now the stack is counted rather than the blocking containers. A good state most likely will have few unordered stacks.

Composite Measure The Composite Measure (CM) is proposed by Scholl et al. [27]. They found that a combination of an underestimate and overestimate function gives decent approximations on the number of reshuffles in the classic CRP. Although this method specifically targets the unloading problem, it may give insights into the combined problem. The exact method of calculation is elaborated in Section 3.2.3.

Batch Label Difference The Batch Label Difference feature (BLD) gives insights into the difference in batch labels among the containers within a stack. A measure around zero indicates a stack of which the batch labels lay close to each other (i.e. these containers depart relatively soon after each other). An extreme value, both positive and negative, indicates a stack where

the batch labels vary a lot. Most likely a good state will have a measure close to zero. BLD is defined as:

$$BLD(L) = \sum_{b \in L} \sum_{s \in L_b} \sum_{i=1}^{H(s)} s_{i-1} - s_i$$

Average Stack Height The Average Stack Height feature (ASH) indicates the average stack height of all non-empty stacks. It is calculated by summing all heights of each stack and divide it by the total number of non-empty stacks. This feature plays a big role in the levelling heuristic. The levelling heuristic has proven to produce competitive results in the setting where the order of retrieval is completely unknown in advance.

Non-Reachable Stacks The Non-Reachable Stacks feature (NRS) indicates the number of stacks that can not be reached. This feature is only applicable to the terminal that uses a reach-stacker, as a gantry crane will have access to every stack at all times. Whether a stack is reachable is determined by if the top-most container of the stack can be retrieved using the reach-stacker stacking restrictions described in Section 5.4. If the topmost container can not be retrieved, the stack is counted as a non-reachable stack. Minimizing the number of non-reachable stacks enforces a way of building where containers are stored in such a way that they remain accessible.

Non-Reachable Containers The Non-Reachable Containers feature (NRC) is an extension to NRS. This feature counts the number of containers that can not be directly reached. If a stack is reachable, the containers below the top-most are counted as not reachable. If a stack is not reachable, all containers in that stack are counted. Minimizing the number of non-reachable containers minimizes the likelihood that a reshuffle is needed to retrieve a container.

Future Blocking Stacks The Future Blocking Stacks feature (FBS) indicates the number of stacks that may become blocking in the future. Whether a stack may become blocking in the future is based on the containers currently residing in the stack and the remaining inbound containers. For every remaining inbound container, the following is checked. First, all containers in the stack that depart earlier than the current inbound container is handled, are removed. Then for the remaining containers, the batch numbers are compared with the inbound container. If an inbound container is departing earlier than one of the remaining containers in the stack, the stack may cause a reshuffle in the future. This stack is then counted for this feature and ignored in the remainder of the calculation (i.e. a stack can only be counted once for this feature). Minimizing the number of future blocking stacks reduces the chance of causing a future blocking stack and thereby the number of reshuffles.

Future Blocking Containers The Future Blocking Containers feature (FBC) indicates the number of containers that may become blocking in the future. Similar to FBS, this feature is calculated by looking at the remaining inbound containers. For every remaining inbound container, it is checked whether it may cause a reshuffle in the future. This is done by first ignoring all containers that depart before the inbound container arrives. The inbound container is counted as future blocking if there is a stack where a container remains that departs earlier than the inbound one. FBC works similar to FBS, only now the inbound containers are counted rather than the stacks.

Apart from the above-mentioned features, also a constant is introduced in every VFA. This constant returns always the same value independent of the state. A benefit of the constant is that for states where all feature functions return zero (for example S_0), a value can be observed. Furthermore, if there is no independent constant in a linear regressions model, the model is forced to go through the origin. This may cause bias in the model. In the remainder of the thesis, the value of the constant is referred to as κ

Results

The above described ADP method is tested using three different runs. The first run serves as a baseline. The second run varies the initial weights and constant used. The third and last run varies the value of delta.

Baseline Figure 6.4 shows the results of the baseline run. The following settings were used for this run: $\theta_t^0 = 1$, $\delta = 0.9$ and $\kappa = 1$. First notable is the rate of convergence. Where the basic ADP solutions still increased in performance and performed rather poorly after 150 iterations, the VFA's perform near-optimal after the same number of iterations.

The performance of the VFA's varies heavily in the first iteration. After the first iteration, VFA1 and VFA4 cause roughly twice the number of reshuffles compared to the other VFA's. These combinations of features, using little tinkered weights, perform more poorly than the other combinations. VFA1 and VFA4 are the only VFA's that use NRC. When looking at the weights, no clear trend could be discovered for this feature. In later iterations, the number of reshuffles is comparable to the other VFA's.

VFA5 seems to perform relatively constant across all iterations, meaning it does not improve over time. VFA5 consists of only three features (as compared to at least four, for the other VFA's). Of these features, NRS yields the same value for all states for terminal 1. NRS does produce distinct values for terminal layout 2, but the same non-learning behaviour is apparent. The other two features (BLB and US) are most likely not sufficient to make a distinction between good and bad states. This is, probably, due to a lot of states yielding the same value, making the distinction between these states hard.

In terms of reshuffles, VFA5 seems to perform the worst and VFA3 the best. The difference between the performance of all VFA's is relatively close. The reason why is probably due to the feature functions being dependent. For example, a high number of unordered stacks means a high number of blocking containers.

When looking at the value estimate of S_0, it can be observed that it does not have the same value as the reported number of reshuffles. This is probably due to the way the weights are updated. In addition, the value of S_0 is solely based on the constant with its updated weights, as all features yield zero. VFA3 and VFA5 even increase in value after the first iteration. This increase means that in the first iteration, these VFA's find a relatively good solution after which in the next iterations poorer are found. An explanation on how this happened is that most likely there are multiple states with the same value. If such a situation occurs, the first one found in a tree-search is selected. This decision yields a semi-random result that performs better than the later iterations.

Weights & Constant The results of the adapted initial weights and constant are shown in Figure 6.5. The same problem instances used in the baseline experiment are run again, but

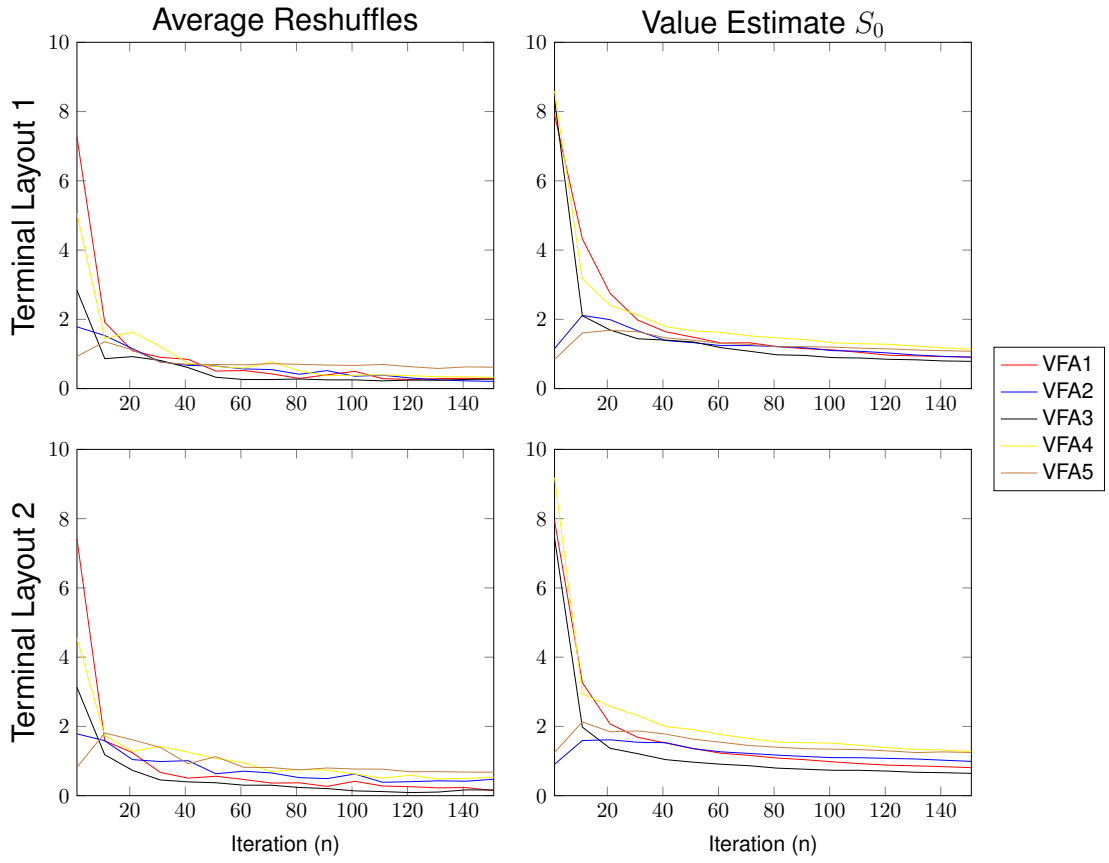

Figure 6.4: Graph showing the results of the five VFA's with settings $\theta_t^0 = 1$, $\delta = 0.9$ and $\kappa = 1$. The two topmost graphs show the results for terminal layout 1 and the two bottom ones for terminal layout 2.

now with $\theta_t^0 = 0$ and $\kappa = 5$. For terminal layout 1, close to no difference in performance with the baseline run can be observed. The initial state value estimate, however, does show a significant difference. In the first iterations, the values are higher than the baseline run. The reason for this is that the constant is higher. This means that after the first iteration the estimate of the initial state is also higher, as the constant is the only factor determining this value.

For terminal layout 2, it can be observed that the algorithm converges slower than the baseline run. This is due to the weights being initialized at zero. As a result, all states yield the same value in the first iteration. A decision, therefore, is made semi-random yielding poor results in the initial iterations. Therefore, it takes longer to find an optimum than when it is already pushed to one by setting the weights bigger than zero. In later iterations, the algorithm does seem to converge and yield similar results as the baseline results. Again, VFA3 yields the best performance, although the margin between VFA1 is smaller.

Delta As running a simulation takes some time and to not clutter the graph, it was decided to alter the different values of δ for a single VFA. VFA3 performed so far the best and therefore chosen to alter different values of δ. Three values for δ were evaluated: $0.1, 0.5$ and 0.9. The remainder of the ADP settings are: $\theta_t^0 = 1$ and $\kappa = 1$. The results are found in Figure 6.6.

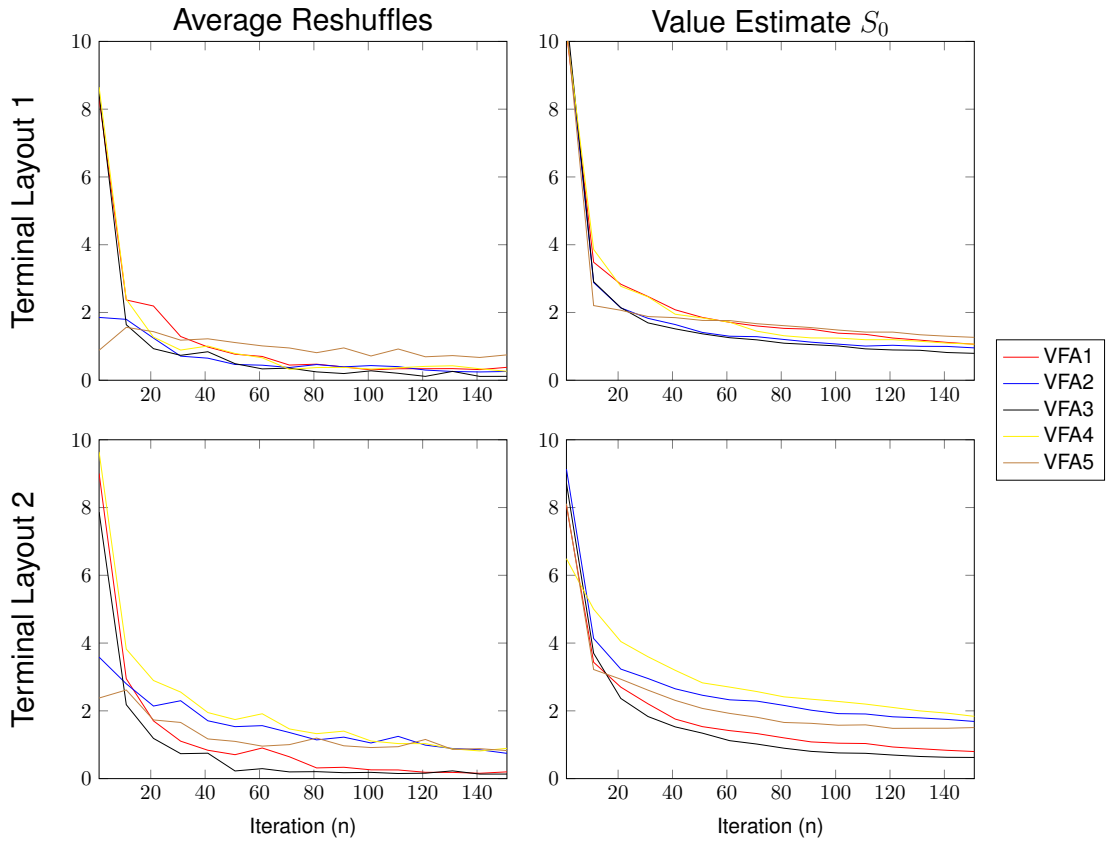

Figure 6.5: Graph showing the results of the five VFA's with settings $\theta_t^0 = 0$, $\delta = 0.9$ and $\kappa = 5$. The two topmost graphs show the results for terminal layout 1 and the two bottom ones for terminal layout 2.

There seems to be little difference between the different values of δ. Most interesting to note is that, in accordance with the theory, a higher value for δ causes the early iterations to have less influence than a lower δ value. In later iterations the effect of δ is negligible. Therefore, the remainder of the experiments is run with the same value of δ as the base-line experiment.

6.3.4 Optimized Basis Function Solution

Even with the restrictions proposed in Section 5.4, the decision space $\mathcal{A}_t$ still grows exponential in terms of the batch and terminal size. Take for instance an inbound batch with two containers and a terminal with 10 stacks operated by a gantry crane. Worst case this would yield $10 * 10 = 100$ possible outcomes. In the current ADP approach, all these outcomes would be generated (the abstraction most likely would cut some outcomes, but cases exist where this is not the case). Therefore, when tackling real-life instances, the ADP approach becomes impractical as the number of possible outcomes is too big to evaluate within a reasonable time. Therefore an optimized ADP approach (in terms of $|\mathcal{A}_t|$) needs to be found.

Let $a_{t,k}$ denote the sequence of container moves for handling the k^{th} container in batch t. This sequence consists of smaller decisions like retrieving a target container, placing a container on a specific stack or reshuffling a container from one stack to another. The latter two of these deci-

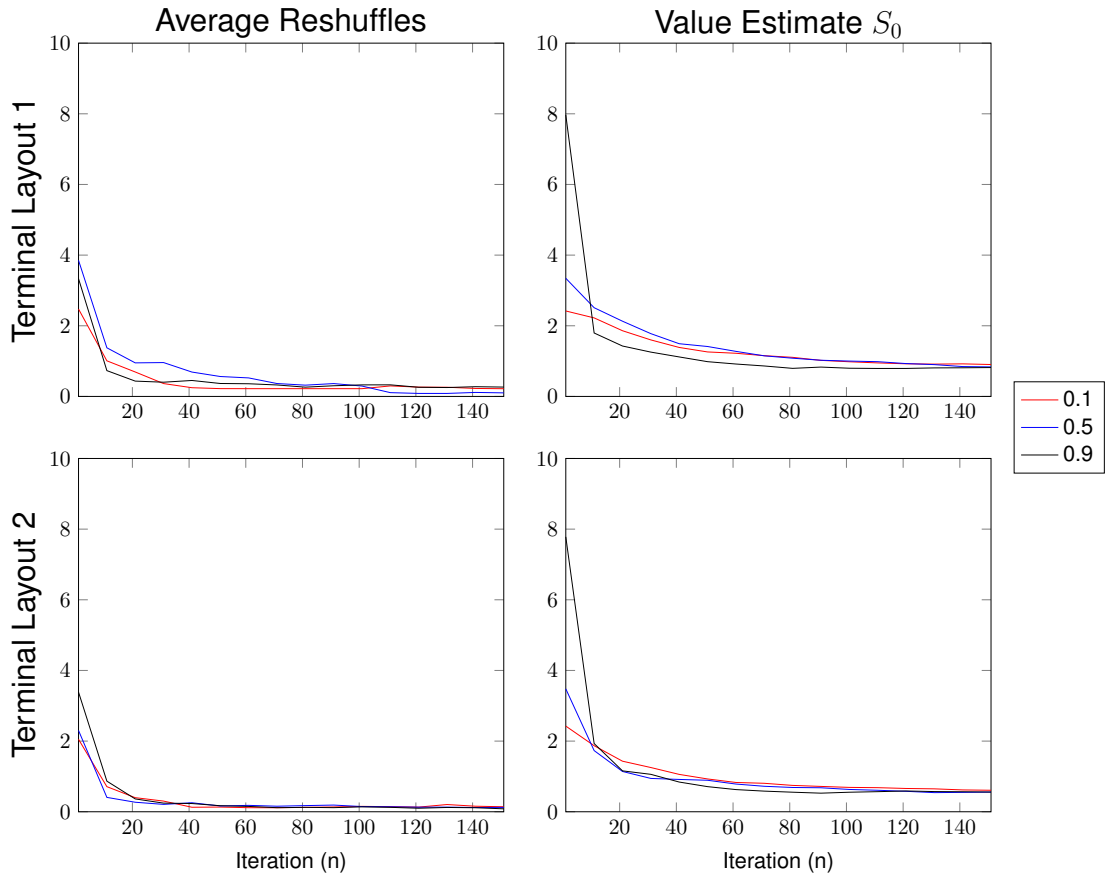

Figure 6.6: Graph showing the results of VFA3 with three different values of δ. The two topmost graphs show the results for terminal layout 1 and the two bottom ones for terminal layout 2.

sions have various possible outcomes. Evaluating a single of such decision is still manageable, even for bigger instances. The problem arises when trying to evaluate all possible outcomes of multiple of these decisions (like the example with two containers). Limiting a smaller decision to only a single possible outcome eliminates the exponential growth.

Limiting the smaller decision to a single decision can be done in various ways. One way would be to limit it to a heuristic. This approach would, however, mean that the ADP approach most likely will not perform better than said heuristic. Therefore it is chosen to use the VFA's. A VFA indicates how good a state is. This can also be applied to the intermediary states between t and $t+1$. Choosing the intermediary state that has the best value according to the VFA yields a single possible outcome per smaller decision (breaking ties by choosing the first discovered). This approach follows the same idea as A^*. A function is used that dictates which state should be expanded next. The major difference is that the proposed approach selects a state, whereas A^* stores all reachable states in a priority queue and explores the most promising one first. This way of deciding has as upside that now the problem scales linear in terms of batch and terminal size, making it, in theory, applicable to real-life instances. A downside of this approach is that it will likely not find an optimum solution for each t. Figure 6.7 shows the results of the optimized approach. To distinguish between the standard and optimized VFA, the optimized VFA's will be

referred to as VFA-O. The same ADP settings as the baseline run are used with the difference that the VFA is used to dictate each smaller decision.

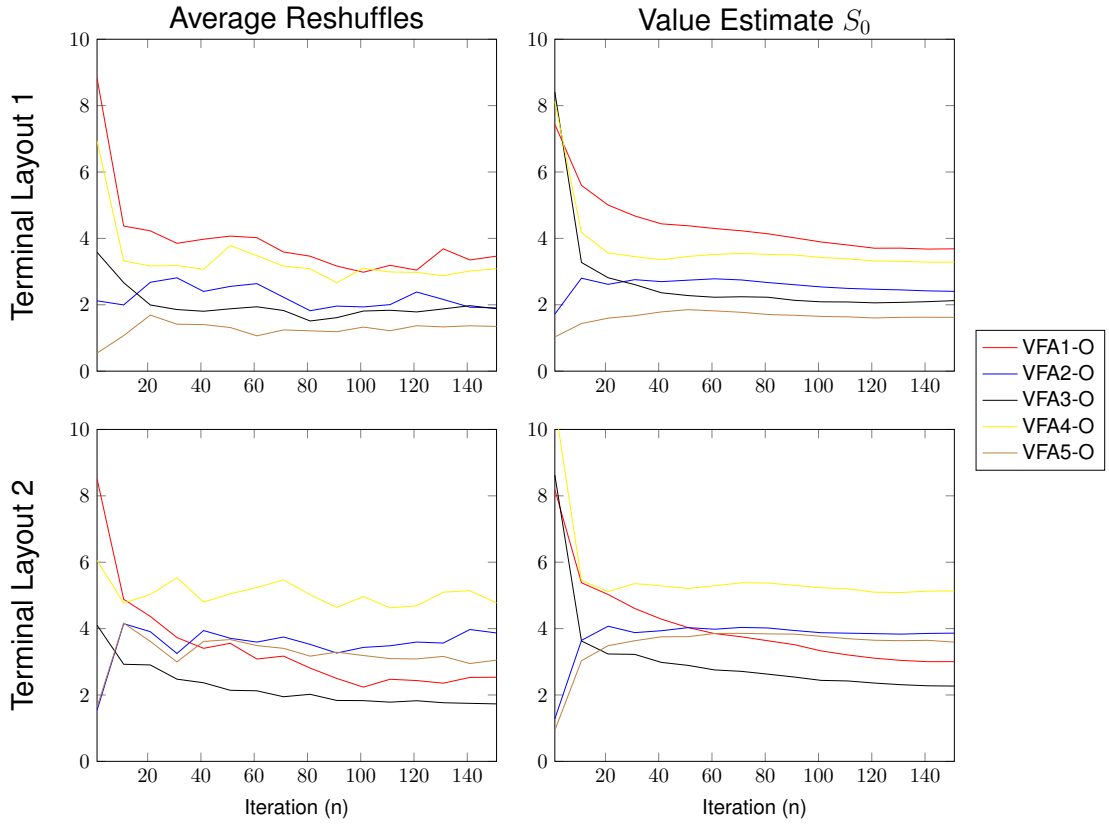

Figure 6.7: Graph showing the results of the five VFA's using the optimized approach. The ADP settings used are $\theta_t^0 = 1$, $\delta = 0.9$ and $\kappa = 1$. The two topmost graphs show the results for terminal layout 1 and the two bottom ones for terminal layout 2.

As can be observed, the performance of the optimized algorithm is way worse than the non-optimized approach. This is expected as another level of approximation is added to the approximation. Similar to the baseline run, the non-learning behaviour can be observed for VFA5. This behaviour is also observed for VFA2. For the other VFA's a slow convergence is observed. A reason why the performance is poor might be simply due to the relatively low number of iterations. Classic ADP approaches run for far more iterations. Another reason might be that the tested VFA's are not suitable to limit the number of decisions. An argument in favour is that the VFA's are not *consistent*. Consistent in this sense means that the estimate is always less or equal then the estimate of a reachable state plus the costs to get there [23]. Formally:

$$\bar{V}_t^n(S_t) \leq C(S_t, a_t) + \bar{V}_{t+1}^n(S_{t+1})$$

Where $\bar{V}_t^n(S_t)$ is the value function approximate for S_t, a_t the decision that leads to S_{t+1} and $C(S_t, a_t)$ the costs of making decision a_t.

6.4 Conclusions

This chapter answered the research question: *"How to apply ADP to the described model?"*. This was done by first introducing ADP to the reader. Afterward, an optimal algorithm was explained to obtain the optimal values of the generated problem instances. Along with the optimal algorithm, an abstraction method was explained. This abstraction method reduces the state space by identifying equivalent states. The abstraction is used to reuse solutions found for equivalent states. Afterward, various ADP approaches were evaluated on two terminal layouts. The first layout represents a terminal operated using gantry cranes and the second represents a terminal operated by reach-stackers. The optimal algorithm could solve all instances of the first terminal layout using zero reshuffles. For the second terminal layout, it could not determine a solution as it ran out of memory.

Of the ADP approaches, first, the single and double pass algorithms were tested. Both used a lookup-table that stored the abstracted states. The algorithms yielded a solution for the first terminal layout, but similar to the optimal algorithm it ran out of memory on the second layout. It was found that both approaches performed poorly and achieved results way off the optimal. The reason for the poor performance is the need for the algorithm to learn all states in order to come up with a good solution. Therefore these approaches are unsuitable for bigger instances. In addition, The costs experienced by the algorithm were not visible in the initial state for the single pass algorithm. Therefore it was chosen to execute the remainder of the experiments with the double pass approach.

Second, the basis function approach was introduced. In total, nine features were implemented and evaluated in different combinations. The weights of the features are updated using the least-squares method for non-stationary data. The basis functions yield a near-optimal solution on the first terminal layout instances. In addition, it performed well, needing one to two reshuffles on average, on the second terminal layout instances. It was found that VFA3 outperformed the other combinations of features, although by a small margin. Altering the weights, the value of the constant and the value of delta did not increase the performance.

Lastly, an optimized basis function approach was proposed. The standard basis function approach can not be applied to bigger instances as the decision space grows too big. The optimized basis function does not evaluate all possible outcomes. It instead uses the basis function to make smaller decisions, thereby eliminating the exponential growth of the problem. The optimized approach, however, did not perform as well as the standard basis function approach. Using the optimized basis function approach, VFA3-O performed best.

In this chapter, it was found that VFA3 performed the best. Both the standard and the optimized version are compared in the next chapter to existing approaches. In turn, the optimized VFA3 will be applied to bigger problem instances, along with existing approaches.

Chapter 7

Comparison

This chapter answers the research question: *"How well does the found ADP solution compare to other approaches?"*. This will be answered by taking the best performing approaches from the previous chapter and compare them with existing approaches found in literature. From the previous chapter, it was found that VFA3 and VFA3-O performed best. The remainder of the chapter is structured as follows. First, the existing approaches are introduced and applied to terminal layout 1 and 2. Afterward, real-life instances are generated and the described approaches are applied to these. All data and code used can be found at: https://github.com/boschma2702/ContainerStacking.

7.1 Comparison Smaller Instances

For comparison, two algorithms are chosen. The first is the Myopic policy. The Myopic policy makes a decision by evaluating all possible outcomes for a given t and selecting the one with the lowest costs. By doing so, it neglects future costs. It was chosen to see whether taking future costs into account has a significant benefit. An explanation of this policy can be found in Section 6.1.1.

The second algorithm is the Min-Max Heuristic (MM) introduced by Caserta et al. [1]. This heuristic bases its decisions on a simple set of rules. The specifics of this algorithm are described in Section 3.2.3. This algorithm is chosen based on its simplicity. In addition, none of its rules conflict with the presence of bays and the restrictions imposed by a reach-stacker. The results of these algorithms, alongside VFA3 and VFA3-O, are shown in Figure 7.1.

As can be observed, the Myopic policy performs significantly worse than the other approaches. This result indicates that it is worthwhile to consider future costs when making decisions. Interesting to note is that the Myopic approach needs fewer reshuffles on terminal layout 2 than terminal layout 1. One would expect to see more reshuffles at terminal layout 2, due to the more limited reachability of containers. It is expected that this is caused by the various stacking restrictions imposed on the problem. These restrictions limit some of the possible bad stack layouts and therefore the algorithm can cause fewer reshuffles. Due to the size of the decision space, the Myopic approach is not applicable to larger instances. Therefore, in combination with its poor performance, it is chosen to not apply this approach to real-life instances.

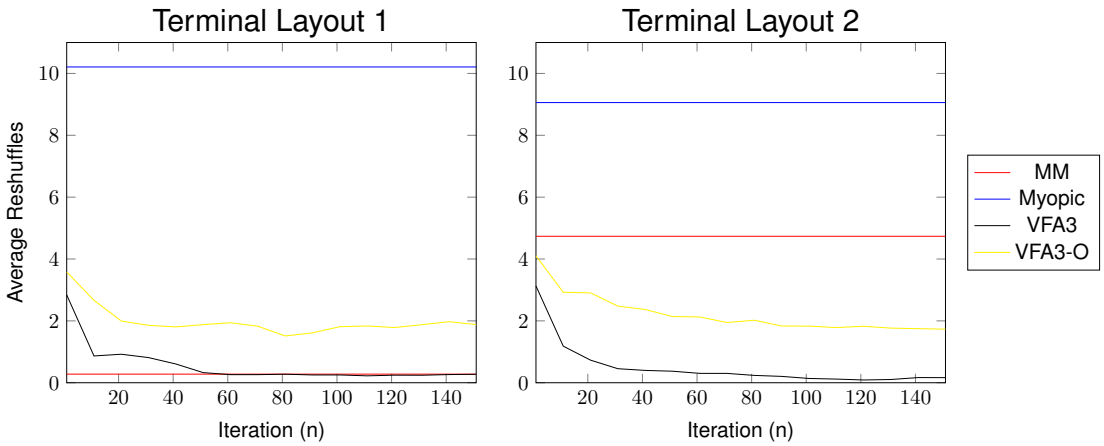

Figure 7.1: Graph showing the average number of reshuffles of the Min-Max Heuristic (MM), Myopic, VFA3 and optimized VFA3 (VFA3-O) on terminal layouts 1 and 2.

MM yields near-optimal solutions for terminal layout 1. However, it is still outperformed by VFA3, although by a small insignificant margin. MM does outperform VFA3-O by quite a bit at terminal layout 1. When looking at terminal layout 2, MM does not yield compelling results. MM is outperformed by both VFA3 and VFA3-O at all iterations. One would expect that after the first iterations a heuristic would outperform an ADP approach as it is not converged yet. An ADP approach would then perform better after a certain number of iterations (i.e. after it has converged). The experiments show that the ADP approach already outperforms the heuristic after the first iteration. Therefore the experiment is run again for VFA3-0, but with weights initialized at zero. The results of this can be found in Appendix A.13. The above-explained behavior is now observed. This indicates that the chosen features already hint toward a decent solution.

Another reason for the ill performance of MM on terminal layout 2 is that the heuristic is not designed for this kind of layout. The first rule states that a stack is chosen, where the earliest departing container of that stack, departs later than the current container being handled. In terminal layout 2 this still, potentially, can result in a reshuffle. Therefore an adapted version of MM is proposed.

Adapted MM-Rule

The first rule of MM currently states that a stack is chosen, where the earliest departing container of that stack, departs later than the current container being handled. This rule narrows the selection of stacks down to ones that do not force a new reshuffle. This rule does not work in a terminal where containers are handled by reach-stackers. In that case, reshuffles might be introduced by neighboring stacks. Therefore a new rule is sought to replace this one. In case a bay is accessible from both sides, it is hard to define a rule that states when a container placed on a specific location will cause a reshuffle or not (i.e. a container may become blocking or unblocked from the other side). In addition, to preserve the simplicity of the algorithm, a simplified rule is used to indicate whether a container becomes blocking.

The Figure 7.2 shows part of a bay. The white square indicates a target location. When a container is placed at this location, it must be accessible from the left or the right (or both). When

this location is accessible from the left, the locations marked by L are checked. A container placed on the target location may become blocking when there exists a container, located on L, that departs earlier. The same check is applied when the location is accessible from the right. Then the locations marked by R are checked. Containers directly below the target location are checked regardless of the direction of accessibility (marked by B).

This rule causes many false positives, meaning that when this rule says a container becomes blocking it might not actually be the case. Therefore, this rule is referred to as potential blocking as the container might not be blocking. The first rule of MM is changed by choosing a stack that introduces no potential blocking container. In the remainder of this thesis, The adapted MM is referred to as MM-A. VFA3 makes use of the CM feature which in turn uses MM. Therefore, an adapted CM feature is created that uses MM-A instead of MM. This change is reflected in an adapted version of VFA3, referred to as VFA3-A. Similarly, an adapted version of VFA3-O is introduced and referred to as VFA3-AO.

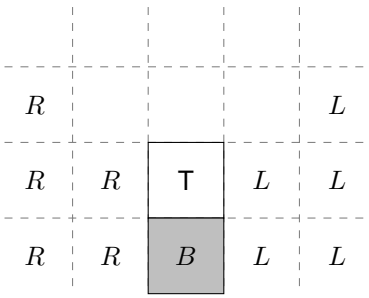

Figure 7.2: Figure showing a bay viewed from the side in 2D. The gray squares represent a container. The white square with a T represents a target location. The letters indicate which locations need to be checked when the target location is accessed from the left (L) or right (R). In addition, the letter B indicates locations that need to be checked regardless of the way of accessing.

The results of the adapted algorithms are shown in Figure 7.3. For terminal layout 1, there is no difference between MM and MM-A. This is expected as the newly introduced rule does not change the decision made by the heuristic when every top-most container is retrievable. A small difference in terms of reshuffles between both VFA3 and VFA3-A can be observed. A similar difference can is present between VFA3-O and VFA3-AO. As the MM-A does not yield a different value for this terminal layout, the difference is explained by the randomness of the ϵ-greedy policy.

For terminal layout 2, MM-A does perform a bit better than MM. Still, all VFA's outperform the heuristic. A similar difference in performance, compared to terminal layout 1, can be observed between the VFA's. Therefore it is concluded that MM-A does not improve the performance compared to MM. The remainder of the experiments will be run using VFA3-O.

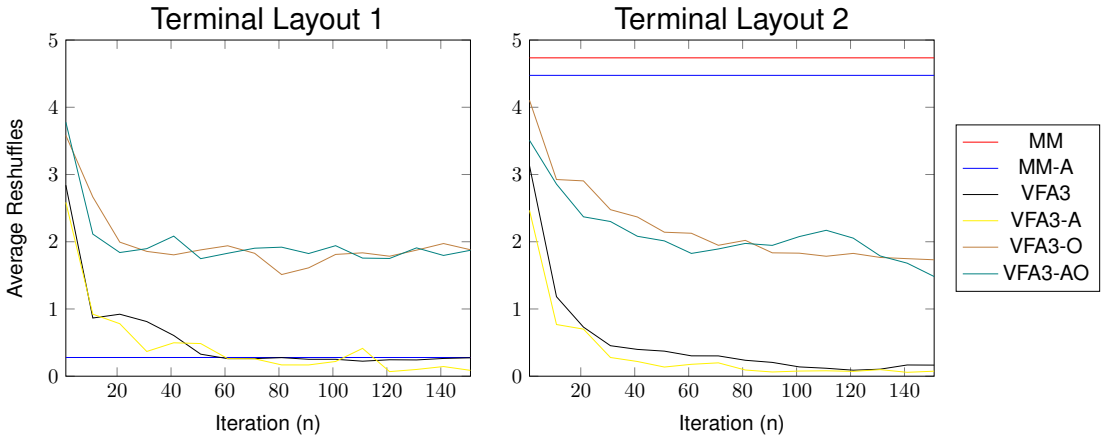

Figure 7.3: Graph showing the average number of reshuffles of the Min-Max Heuristic (MM), the adapted MM (MM-A), VFA3, VFA3 using MM-A (VFA3-A), optimized VFA3 (VFA3-O) and optimized VFA3 using MM-A (VFA3-AO) on terminal layouts 1 and 2.

7.2 Real-Life Instances

From Chapter 4 it is concluded that there are two classes of clients: clients who handle on average no more than 500 containers a week and clients that handle on average 1000 a week. Based on these numbers, four problem instances are generated. The problem instances consist of 40 time periods. These periods are based on 5 working days a week and an 8-hour working day. The four problem instances consist out of, respectively, 100, 250, 500 and 1000 inbound containers. For convenience, the problem instances are given the names: *tiny*, *small*, *medium* and *big* respectively. Each inbound container has an average dwell time of 15 time periods (equals 3 days). The same method for generating the smaller problem instances explained in Section 6.3.1 is used.

Each problem instance is tested against its own terminal layout. These layouts are based on findings from Chapter 4. Roughly on average, a terminal had 1 stack per inbound container per week. The same ratio is used when creating the terminal layouts. Therefore, the number of inbound containers in a problem instance also dictates the number of available stacks. Furthermore, all analyzed clients operate the terminal using reach-stackers. Based on satellite images, it was determined that the most common bay size is five stacks and that they are accessible from both sides. For simplicity, it is assumed that all bays consists of five stacks and are accessible from both sides. This results in 20, 50, 100 and 200 bays respectfully for the real-life problem instances.

The algorithms MM, MM-A and VFA3-O are applied to all problem instances. The results are discussed per problem instance. During experimentation, it was found that MM and MM-A yielded results for all real-life instances within a reasonable time, but VFA3-O, apart from the tiny instance, not. Therefore two measures are introduced that guarantee a faster termination.

The first measure targets the number of evaluation runs. Currently, a major part of the calculation time involves running these simulations. As the focus is on how well the algorithm performs and not how it converges, it is decided to only run an evaluation after the last learning iteration. The second measure targets the remaining curse of dimensionality that is still haunting;

the decision space. To further reduce this, the corridor method of Caserta et al. [4] is adapted and used. The next section will elaborate on the adapted corridor method. After describing the corridor method, the results of the real-life instances are given.

7.2.1 Corridor Method

The corridor method is proposed by Caserta et al. [4]. This method limits the decision space by only allowing reshuffles to neighboring stacks. From a practical view, it makes sense to include such a restriction, as it decreases the travel time during a reshuffle. Caserta et al. applies the method to the CRP. In this thesis, the approach is adapted to serve the combined problem and work with bays.

First, the corridor itself is introduced. Only a horizontal corridor is used and no vertical. The reason being that stacking in bays already imposes height restrictions on neighboring stacks. In the original method, the stacks are counted as neighbors. In this thesis, it is decided to count bays as neighbors. The corridor thus forms a range of bays rather than a range of stacks. Only bays within the range are considered when making a decision. The middle of the range is the bay from which a blocking container is reshuffled. The value of the corridor indicates which bay, seen from the middle, is included. *Example:* A corridor value of two indicates that in total five bays are considered.

Applying the corridor method to reshuffles decreases the number of bays that need to be considered. Ideally, the same is done for inbound containers. The problem, however, is that inbound containers do not have a bay from which they need to be reshuffled. Therefore, it is unknown where to place the corridor around. To solve this, it is chosen to draw a random bay index (uniformly distributed). Around the drawn index, the corridor is set. This way, when placing a container, both reshuffle and inbound, only a small number of bays are considered. *Example:* Only five bays are considered when using a corridor of two, compared to the 200 possible ones for the big instance.

Currently, the features of the basis function still consider the entire terminal. Most features can be split into two parts. The first part calculates the value of the bays within the corridor and the second part outside. An example of this is the calculation for the Blocking Lower Bound (BLB). The number of blocking containers in the terminal is equal to the blocking containers within the corridor plus the number of blocking containers outside the corridor. As only changes within the corridor are allowed, it is known that the value outside the corridor remains constant across all considered decisions. Therefore, one only needs to calculate the value within the corridor to get the best-valued state.

The Composite Measure (CM) is the only feature that does not adhere to the independent calculation. Simultaneously, it is the measure that takes the most time to compute. Therefore an adaptation to this feature is introduced. When using a corridor, only containers within that corridor are considered. CM now calculates the value as if the corridor was a terminal on its own. All containers and available locations outside of the corridor are ignored.

Next, the corridor method is incorporated in VFA3-O and tested against the real-life instances. A corridor size of one and two are tested and respectively referred to as VFA3-OC1 and VFA3-OC2.

7.2.2 Results

This section discusses the results of the identified algorithms on the different real-life instances. The calculation times of the tiny and small instances were small enough to include a graph of convergence. VFA3-O even could be applied to the tiny instance.

Tiny

The tiny terminal layout consists out of 20 bays, each with five stacks. In total, 100 containers arrive during the planning horizon. With that number of containers, it is smaller than all clients. It comes most close to Client D, which handles around 150 containers per week. The result of the tiny instance can be found in Figure 7.4. VFA3-O outperformed the heuristic significantly. Even after a single iteration, the algorithm already causes fewer reshuffles than the heuristics.

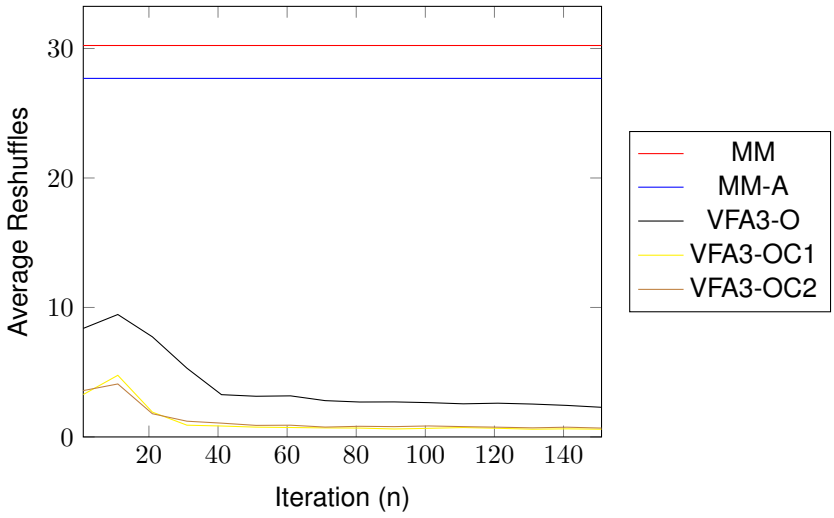

Figure 7.4: Graph showing the average number of reshuffles of the Min-Max Heuristic (MM), the adapted MM (MM-A), optimized VFA3 (VFA3-O), optimized VFA3 using a corridor of 1 (VFA3-OC1) and optimized VFA3 using a corridor of 2 (VFA3-OC2) on the tiny problem instance.

An interesting observation is that VFA3-OC1 and VFA3-OC2 outperform VFA3-O. One would expect that these algorithms perform worse as the number of possible decisions is reduced by the corridor. One argument why both VFA3-OC's outperform VFA3-O could be the random corridor selection for inbound containers. It could be that the random corridor cuts off bad decisions. This randomness, however, most likely would not yield consequent better results averaged over 16 problem instances. Our hypothesis is that the random corridor forces the algorithm to make more random decisions than the original. The corridor is likely to cut off an, what the algorithm currently thinks, optimal solution at one point. Then the algorithm is forced to pick another decision which may yield a better solution than what it thought was best. This hypothesis might be confirmed by running the experiments of the previous chapter, altering the value for epsilon.

Furthermore, after ten iterations a bump can be observed in the number of reshuffles. The algorithm performs worst after ten iterations. One would expect the worst performance after the first iteration. This behavior is also present at VFA3-O, while it did not occur at terminal layout 1

and 2. No clear explanation for the bump could be given.

Small

The small terminal layout consists out of 50 bays, each with five stacks. In total, 250 containers arrive during the planning horizon. Thereby it handles little under the number of containers handled by Client B (300) and about the same as Clients C and F. The results on the small instances can be found in Figure 7.5. As can be observed, both heuristics perform worse than the ADP algorithms. Compared to the tiny real-life instances, a larger difference in the number of reshuffles can be observed (both absolute and relative). It seems that the bigger the problem grows, the worse MM performs compared to MM-A.

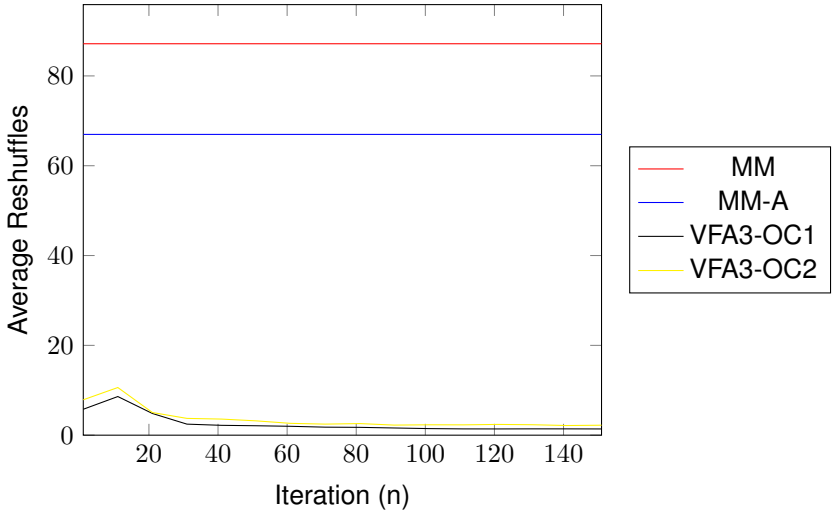

Figure 7.5: Graph showing the average number of reshuffles of the Min-Max Heuristic (MM), the adapted MM (MM-A), optimized VFA3 using a corridor of 1 (VFA3-OC1) and optimized VFA3 using a corridor of 2 (VFA3-OC2) on the small problem instance.

VFA3-O could not be computed for this instance. Therefore, this approach is ignored for the remainder of real-life instances. The VFA3-OC's outperform both heuristics and yield similar results as to the tiny instances. After the final iteration, the algorithm yields a bit over twice the number of reshuffles compared to the tiny instance (1.39 and 2.22 vs 0.60 and 0.69).

When compared to the reported reshuffles of Clients C and F, it can be noted that on average the ADP algorithm yields fewer reshuffles. It, however, must be noted that the ratio of available stacks versus containers arriving in a week is higher at Client C and lower at Client F. Furthermore, the evaluation runs with perfect information (apart from the order within a batch), whereas in real life this is often not the case.

Medium

The medium terminal layout consists out of 100 bays, each with five stacks. In total, 500 containers arrive during the planning horizon. Thereby it handles little over the peak weeks of Clients C and B. MM yielded 188.03 reshuffles and MM-A 126.33. Again both absolute and relative differences increased.

The computation times for this terminal layout were too long to include an evaluation of every 10^{th} iteration. Therefore, only an evaluation is run after the final iteration. VFA3-OC1 yielded 3.14 reshuffles and VFA3-OC2 resulted in 3.70 reshuffles. Thereby, it again outperforms both heuristics. Compared to the small real-life instances, again, a little over twice the number of reshuffles is observed.

Big

The big terminal layout consists out of 200 bays, each with five stacks. In total, 1000 containers arrive during the planning horizon. Thereby it handles around the same number of containers as Clients A, E and G. MM yielded 370.20 reshuffles and MM-A in 229.74. Similar to the previous instances, an increase in difference is observed. However, the relative increase gets smaller and smaller when the problem size increases.

VFA3-OC1 resulted in 6.04 reshuffles and VFA3-OC2 in 6.12. It, again, outperforms both heuristics. When compared to the number of reported reshuffles of Clients E and G, on average, the number of reshuffles achieved by the ADP approaches is lower. However, there is a high variability in these numbers, so a clear cut comparison is hard. In addition, the ratio of available stacks is much lower than tested.

Discussion

In this section, two approaches from literature were proposed. Both approaches were outperformed by the found ADP solutions. However, this comparison may not be that fair. The first approach, the Myopic, was to be expected to perform worse as it only takes direct costs into account. The second approach, MM, is a heuristic that is designed for a terminal operated by gantry cranes and performed well on these instances. It performed mediocre on the terminal instances operated by reach-stackers. A simple explanation for this performance is that the heuristic is not designed to be applied to such layouts, resulting in a not so fair comparison. A well-designed heuristic might yield competitive or even better results. To the best of our knowledge, no literature exists that explores a heuristic that is defined for a terminal operated using reach-stackers. Therefore, an adaptation to the MM Heuristic was proposed. This adaptation, however, is an oversimplified rule on when a container becomes blocking in a bay. Thereby, it will disregard various locations that might not actually yield a reshuffle, resulting in a policy that could perhaps produce better solutions with an improved adaptation.

Next, this section described the generation of real-life instances. During testing of the algorithms, the terminal starts empty. This does not reflect real-life practices as there are still containers left from previous weeks. Also, these instances are tested using the entire planning horizon (i.e. a week). Knowing a week in advance for all containers roughly when they arrive and depart is highly unlikely in real-life operations. Both have the effect to cause a lower number of reshuffles than in real-life. Ideally, one wants to test the algorithms as close to real-life as possible. This can be achieved by first having a week in which the algorithms prepare the terminal. Simulating multiple realizations will result in multiple starting points for the week in which the algorithms are compared against each other. However, this approach, due to the execution time of the ADP algorithms, is unfeasible. Lowering the planning horizon would decrease the execution times, as well as reflecting real-life a bit better. However, it is unsure to which extend terminals know in advance when containers roughly arrive and depart. As a compromise, it was decided to use the described method of evaluation. By extending the planning horizon, the effects of starting with an empty terminal are a bit diminished as during the horizon containers

already depart while new ones still arrive. In addition, by using a wider planning horizon, it shows what can be achieved if information is known more in advance.

Every real-life instance has its own terminal layout. For simplicity, it was decided that all bays consists of five stacks and is reachable from both sides. In reality, this may be not the case and bays of different sizes and reachability may be present within a single terminal. Besides, it is assumed that every client has the same number of stacks per container that arrives in a week. This number was derived from the reported used stacks from clients. This number varied between clients. For instance, Clients C and F handle roughly the same number of containers as the small instance. Client C has more than four times more stacks available than Client F. This, in addition to the above-mentioned differences between real-life, makes a comparison between the described results in this thesis and the reported number of reshuffles hard. Therefore, the comparison made between these two should be taken with a grain of salt.

Furthermore, the method used to determine the performance of the algorithms can be improved. Each experiment solves the problem for 16 instances. This way, the algorithms are tested against a broader set of problems and diminishes the possibility of finding an algorithm that only performs well on a specific problem instance. In addition, each experiment is only run once. This might lead to that randomness plays a role in the outcomes and that the best-found approaches were just lucky during evaluation. As approaches are applied to 16 instances and the averages are reported, this effect is somewhat diminished. Running more problem instances multiple times results in more certainty in the performance of the algorithms.

7.3 Conclusions

This chapter answered the research question *"How well does the found ADP solution compare to other approaches?"*. This was achieved by taking the best found ADP algorithm described in the previous chapter and compare them with existing approaches in literature. The Myopic and Min-Max Heuristic (MM) were chosen to compare to VFA3 and VFA3-O. These approaches were applied to terminal layout 1 and 2 and compared to the results of the previous chapter. It was found that the Myopic approach had, by far, the worst performance. This performance indicates that it is worth to consider future costs. MM produces near-optimal results for terminal layout 1 but performs mediocre on terminal layout 2. It was thought that this performance was caused by the first rule of MM, which does not precisely apply to terminals operated by reach-stackers. Therefore, the first rule of MM was adapted to make use of potential blocking containers. The altered heuristic is referred to as MM-A. MM-A was found to perform marginally better than MM on terminal layout 2. As MM is part of one of the features of VFA3, VFA3-A was introduced that makes use of MM-A. The same holds for VFA3-O for which VFA3-AO was introduced. It was found that the adaptation does not yield a significant increase in performance for the ADP approaches. Furthermore, it was found that VFA3 yields better results on terminal layout 1 than MM, although the margin is insignificant. VFA3, however, performs significantly better at terminal layout 2 than both MM and MM-A. VFA3-O performed worse at terminal layout 1 than MM. It did outperform both MM and MM-A on terminal layout 2.

Next, four real-life problem instances referred to as tiny, small, medium, and big were generated. These instances are generated using the same technique as for terminal layout 1 and 2. Due to the nature of the Myopic approach and its ill performance, it was decided to not apply this approach to the real-life instances. In addition, VFA3, as discussed in the previous chapter, can not be applied to the real-life instances due to the exponential growth of the decision space. During experimentation on these problem instances, it was found that the optimized

basis function approach took too long to yield results. Only results for the tiny instance could be obtained. Therefore two measures were taken to obtain results in a reasonable time. The first measure is the elimination of most of the evaluation simulations for the medium and the big instances. On these instances, only an evaluation after the last learning iteration is performed. The second measure is applying the corridor method. The corridor method limits the allowed decisions by only allowing container moves to be carried out to a certain neighborhood of bays. Corridor sizes of 1 and 2 were tested. The corridor algorithms are referred to as VFA3-OC1 and VFA3-OC2.

In all real-life instances, the ADP approaches outperformed the heuristics. It is observed that when increasing the problem size, the number of reshuffles needed grows faster for the heuristic than the tested ADP approaches. The VFA3-OC's outperform VFA3-O. This behavior is odd, as one would expect that by having more choices available, a better solution should be achievable. The contrary is true when looking at the experiments. Although this finding is only based on a single experiment, our hypothesis is that by making use of a smaller corridor, the algorithm is forced into a more explorative behavior, resulting in better overall performance. Results even seem to suggest that a corridor of 1 outperforms a corridor of 2. The evidence of the latter, however, is not convincing as the difference in performance is small and insignificant. More study is needed on the difference in the performance of different corridor sizes.

Chapter 8

Conclusion

The goal of this research was to find an approach that determines real-life applicable container stack allocations that minimize the number of reshuffles. In order to achieve the research goal, five research questions were formulated and answered. The first research questions analyzed various related works and found different variations of the problem and various approaches to solving them. In addition, a classification scheme was found and identified work was classified using this scheme.

The second research question analyzed various characteristics of day-to-day container handling. This was achieved by investigating the data of seven clients of Cofano. It was found that the clients heavily varied in the number of containers handled per week. Three clients handled around 1000 inbound containers per week, whereas the others handled no more than 500. On average, containers remain three days at the terminal.

The third research question proposed the model used in the remainder of the thesis. It first introduced the basic concepts of SDP. Afterward, eight assumptions were elaborated and defended after which the model was defined. In addition, restrictions imposed by the use of a reach-stacker were elaborated. Furthermore, a method to use the defined model for both a terminal operated by reach-stackers, as well as gantry cranes, was introduced.

The fourth research question described the method to apply ADP to the proposed model. It first introduced the basic concepts of ADP. Afterward, an optimal policy was introduced which yields solutions based on minimizing the expected number of reshuffles. This policy was based on the PBFS algorithm of Galle et al. [10] and made use of an abstraction technique that reduces the number of possible states which need to be explored. Next, various ADP approaches were evaluated on two terminal layouts. The first terminal layout represents a terminal operated by gantry cranes and the second, a terminal operated by reach-stackers. On both terminal layouts the ADP algorithm using a lookup table storing the abstracted states were tested. This algorithm was tested using the both the single and the double pass approach. Both variants of the algorithm did not yield compelling results on the first terminal layout and ran out of memory on the second layout. In addition, the costs experienced by the single pass approach were not reflected back to the initial state. Therefore it was decided to use the double pass approach for the remainder of the thesis. Subsequently, the basis function approach was introduced. In total, nine features were defined and five different combinations were tested. It was found that VFA3 performed best. It yielded near-optimal solutions for the first terminal layout and yielded

promising results for terminal layout 2. Due to the exponential growth of the decision space, the basis function approach is not suitable for real-life instances. Therefore an optimized basis function was proposed. It was found that the optimized version of VFA3 (referred to as VFA3-O) performed the best of these approaches. VFA3-O did perform less than VFA3.

The fifth and final research question compared the best-found approaches to existing ones in literature. It was decided to use the Myopic policy and the Min-Max Heuristic (MM) (proposed by Caserta et al. [1]). First, these approaches were applied to the same problem instances as the ADP approaches. It was found that the Myopic policy had the worst performance on both terminal layouts. This result suggests that it is worth to take future costs into account. MM yielded near-optimal solutions for the first terminal layout but performed mediocre on the second terminal layout. It was thought that this performance might be due to the nature of the first rule of the heuristic. The first rule states that a stack is chosen, where the earliest departing container of that stack, departs later than the current container being handled. Therefore this rule was adapted to take blocking containers within the same bay into account. This adapted version of MM is referred to as MM-A. It was found that MM-A resulted in slightly better solutions for terminal layout 2. MM-A is still outperformed by both VFA3 and VFA-O. Next, four real-life problem instances were defined: *tiny*, *small*, *medium* and *big*. These instances span a week and contain respectively 100, 250, 500 and 1000 inbound containers. It was found that VFA3-O could not yield results in a reasonable time. Therefore two measures were introduced. The first measure eliminates, apart from the final iteration, all evaluation simulations for the medium and big instances. The second measure is introducing the corridor method. The remainder of the experiments were executed using VFA3-O with corridor sizes of one and two (referred to as VFA3-OC1 and VFA3-OC2 respectively). On the tiny instances, both VFA3-OC's outperformed VFA3-O. The hypothesis is that the corridor forces the algorithm to make more random decisions. This random behavior results in more explorative behavior and results in overall better solutions. Furthermore, it was found that the VFA3-OC's significantly outperformed the heuristics on all real-life instances. Lastly, no clear conclusions could be drawn in comparison to the reported reshuffles of clients. All problem instances and code used to achieve the results in this thesis can be found at: `https://github.com/boschma2702/ContainerStacking`.

8.1 Future Work

In this work, it is shown that the optimized basis function approach in combination with the corridor method is a promising approach for tackling the stacking problem. It remains, however, if this statement still holds when applied in real-life. Part of this uncertainty is due to the inherent nature of abstracting from a real-life process. Issues present in the evaluation of the algorithms are already discussed in the previous chapter. Issues present in the model itself remain to be solved. An example of such an issue is the assumption that for a given planning horizon, all containers are known. In reality, this may not be the case and containers might arrive out of their 'batch' or additional ones appear. The current model does not allow for such events and this needs to be addressed before the model can be optimally used in a real-life setting. In addition, the model assumes that all containers are the same. In reality, this is not the case. Stacking restrictions may be imposed on the type of container or based on its contents. Also, the notion of container families exists. An example of a family are the empty containers. When retrieving a container, it does not matter which specific one is chosen, as long as it is one of the family. This increases the complexity of the problem but allows for more flexibility. All these additions to the model are possible to address in future work.

Another uncertainty that remains is the execution time of the algorithms. All real-life results could be obtained within a day[1]. These results could only be achieved by heavily restricting the decision space. It would be interesting to see whether a decrease in the number of reshuffles can be achieved by alleviating some of these restrictions. We believe that the most promising would be the basis function in combination with the corridor method, leaving out the optimized part. This can only be evaluated in a reasonable time when execution efficiency issues are addressed for the bigger problem instances. Currently, 16 problem instances are solved in parallel. In real life, only a single problem instance per client needs to be solved. This brings the opportunity to solve a single problem in parallel. A single problem can be divided into two or more smaller problems which can be solved separately and in parallel. This approach reduces the calculation time drastically. It, however, remains unclear how to split the problem. Besides, currently, the corridor method chooses a random bay when deciding where to place an inbound container. It should be able for the algorithm to learn which bay is most promising by aggregating states in the value function or learning a new value function altogether.

Another way to decrease the computation time is to implement a more efficient way of simulating. This can be achieved by altering the implementation of the logic of the simulation. A different solution would be to implement the algorithms in a lower-level language such as C, Go or Rust. The decrease in computation time gained from these changes may already be enough to evaluate a bigger decision space.

In this thesis, the problem is modeled as a finite horizon problem. It would be interesting to see how a different way of modeling would compete with the one described in this thesis. The weights in a finite horizon problem need to be learned again and again for each problem instance. An infinite horizon problem does not have this limitation, as only a single set of weights is learned. This, in turn, can be applied to all problem instances where the underlying probability distributions are the same. This approach, however, would most likely perform worse than the finite horizon version as less information is known during planning. In addition, the weights are tweaked to perform well overall stages rather than a specific t. On the other hand, one needs to calculate the weights only once and as long as the underlying distribution of a client remains unchanged, it can be reused. This allows one to run more learning iterations, perhaps compensating the performance this way.

Lastly, it would be interesting to explore the impact on the performance of different features. In this thesis, only nine features were introduced. This number is incredibly small compared to the number of possible states in the model. One could introduce more distinct features to describe a state. A way to accomplish this is to introduce a feature per stack or bay. This way the number of features included grows with the size of the terminal.

[1] All experiments were run on a computing cluster. No conclusions can be drawn from the execution times as instances run on different underlying hardware and calculations may be temporary suspended.

Bibliography

[1] Caserta, M., Schwarze, S., and Voß, S. (2009). A New Binary Description of the Blocks Relocation Problem and Benefits in a Look Ahead Heuristic. European C:37–48.

[2] Caserta, M., Schwarze, S., and Voss, S. (2011a). Container Rehandling at Maritime Container Terminals. *Handbook of terminal planning*, 49(1):247–269.

[3] Caserta, M., Schwarze, S., and Voß, S. (2012). A mathematical formulation and complexity considerations for the blocks relocation problem. *European Journal of Operational Research*, 219(1):96–104.

[4] Caserta, M., Voß, S., and Sniedovich, M. (2011b). Applying the corridor method to a blocks relocation problem. *OR Spectrum*, 33(4):915–929.

[5] Conger (n.d.). Toyota Reach Stacker Container Handler. https://www.conger.com/product/toyota-reach-stacker-container-handler/. [Online; accessed 2020-03-03].

[6] Corvus Energy (2015). Rubber Tired Gantry (RTG) Crane hybridation with Corvus Energy ESS. https://corvusenergy.com/two-new-hybrid-rubber-tired-gantry-cranes-rtgs-powered-by-corvus-energy-to-be-installed-in-yidong-terminal-port-of-shanghai/. [Online; accessed 2020-03-03].

[7] Dekker, R., Voogd, P., and Van Asperen, E. (2006). Advanced methods for container stacking. *OR Spectrum*, 28(4):563–586.

[8] Economic Commission for Europe Committee on Inland Transport (2014). European agreement concerning the international carriage of dangerous goods by road. Technical report, UNITED NATIONS.

[9] Galle, V., Barnhart, C., and Jaillet, P. (2018a). A new binary formulation of the restricted Container Relocation Problem based on a binary encoding of configurations. *European Journal of Operational Research*, 267(2):467–477.

[10] Galle, V., Manshadi, V. H., Boroujeni, S. B., Barnhart, C., and Jaillet, P. (2018b). The stochastic container relocation problem. *Transportation Science*, 52(5):1035–1058.

[11] Gambardella, L., Bontempi, G., Taillard, E., Romanengo, D., Raso, G., and Piermari, P. (1996). Simulation and forecasting in intermodal container terminal. *Proceedings of the 8th European Simulation Symposium*, pages 626–630.

[12] Gharehgozli, A. H., Yu, Y., De Koster, R., and Udding, J. T. (2014). A decision-tree stacking heuristic minimising the expected number of reshuffles at a container terminal. *International Journal of Production Research*, 52(9):2592–2611.

[13] Güven, C. and Türsel Eliiyi, D. (2019). Modelling and optimisation of online container stacking with operational constraints. *Maritime Policy and Management*, 46(2):201–216.

[14] ISO/TC 104/SC 4 (1995). Iso 6346:1995 freight containers coding, identification and marking.

[15] Kim, K. H. and Hong, G. P. (2006). A heuristic rule for relocating blocks. *Computers and Operations Research*, 33(4):940–954.

[16] Kim, K. H. and Park, K. T. (2003). A note on a dynamic space-allocation method for outbound containers. *European Journal of Operational Research*, 148(1):92–101.

[17] Kim, K. H., Park, Y. M., and Ryu, K.-R. (2000). Deriving decision rules to locate export containers in container yards. *European Journal of Operational Research*, 124(1):89–101.

[18] Ku, D. and Arthanari, T. S. (2016). Container relocation problem with time windows for container departure. *European Journal of Operational Research*, 252(3):1031–1039.

[19] Lehnfeld, J. and Knust, S. (2014). Loading, unloading and premarshalling of stacks in storage areas: Survey and classification. *European Journal of Operational Research*, 239(2):297–312.

[20] Li, W., Xiaoning, Z., and Zhengyu, X. (2019). Efficient container stacking approach to improve handling: efficiency in Chinese rail–truck transshipment terminals. *Simulation*, 96(3):3–15.

[21] Mes, M. R. and Pérez Rivera, A. (2017). Approximate dynamic programming by practical examples. *International Series in Operations Research and Management Science*, 248(February):63–101.

[22] Murty, K. G., Wan, Y.-w., Liu, J., Tseng, M. M., Leung, E., Lai, K.-K., and Chiu, H. W. (2005). Hongkong international terminals gains elastic capacity using a data-intensive decision-support system. *Interfaces*, 35(1):61–75.

[23] Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., USA.

[24] Petering, M. E. H. and Hussein, M. I. (2013). A new mixed integer program and extended look-ahead heuristic algorithm for the block relocation problem. *European Journal of Operational Research*, 231(1):120–130.

[25] Powell, W. B. (2013). *Approximate Dynamic Programming*. John Wiley & Sons.

[26] Ries, J., González-Ramírez, R. G., and Miranda, P. (2014). A fuzzy logic model for the container stacking problem at container terminals. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8760:93–111.

[27] Scholl, J., Boywitz, D., and Boysen, N. (2018). On the quality of simple measures predicting block relocations in container yards. *International Journal of Production Research*, 56(1-2):60–71.

[28] Tanaka, S. and Takii, K. (2016). A faster branch-and-bound algorithm for the block relocation problem. *IEEE Transactions on Automation Science and Engineering*, 13(1):181–190.

[29] Tang, L., Jiang, W., Liu, J., and Dong, Y. (2015). Research into container reshuffling and stacking problems in container terminal yards. *IIE Transactions (Institute of Industrial Engineers)*, 47(7):751–766.

[30] Techniques, C. (2009). An introductory guide to port cranes and the application of variable speed drives. Technical report.

[31] UNCTAD (2018a). Container port throughput, annual. https://unctadstat.unctad.org/wds/TableViewer/tableView.aspx. Accessed: 2020-02-07.

[32] UNCTAD (2018b). Review of maritime transport 2018. Technical report, UNITED NATIONS CONFERENCE ON TRADE AND DEVELOPMENT.

[33] Ursela Horn (2009). Integral reefer container. https://commons.wikimedia.org/wiki/File:Lauritzen1_k100.jpg. [Online; accessed 2020-03-03].

[34] Wan, Y. W., Liu, J., and Tsai, P.-C. (2009). The Assignment of Storage Locations to Containers for a Container Stack. *Naval Research Logistics*, 56(8):699–713.

[35] Wu, K.-C. and Ting, C.-J. (2010). A beam search algorithm for minimizing reshuffle operations at container yards. In *Proceedings of the international conference on logistics and maritime systems*, pages 15–17.

[36] Zeng, Q., Feng, Y., and Yang, Z. (2019). Integrated optimization of pickup sequence and container rehandling based on partial truck arrival information. *Computers and Industrial Engineering*, 127(August 2018):366–382.

[37] Zhang, C., Chen, W., Shi, L., and Zheng, L. (2010). A note on deriving decision rules to locate export containers in container yards. *European Journal of Operational Research*, 205(2):483–485.

[38] Zhang, C., Wu, T., Kim, K. H., and Miao, L. (2014). Conservative allocation models for outbound containers in container terminals. *European Journal of Operational Research*, 238(1):155–165.

[39] Zhu, W., Qin, H., Lim, A., and Zhang, H. (2012). Iterative deepening a* algorithms for the container relocation problem. *IEEE Transactions on Automation Science and Engineering*, 9(4):710–722.

Appendices

A.1 Inbound Containers per Modality

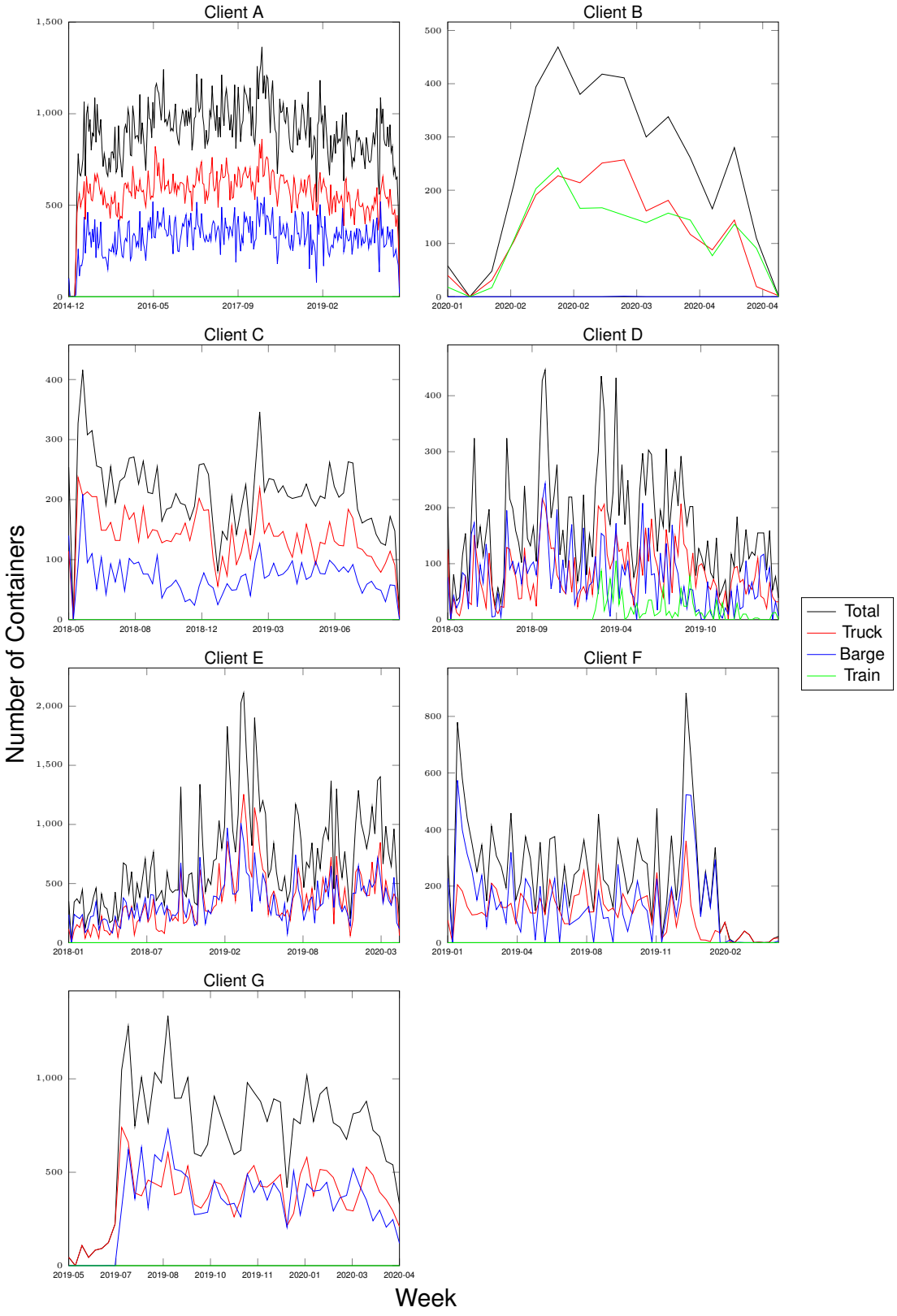

Figure 1: Overview of the number of containers handled per week. The colour indicates via which modality the containers arrive.

A.2 Outbound Containers per Modality

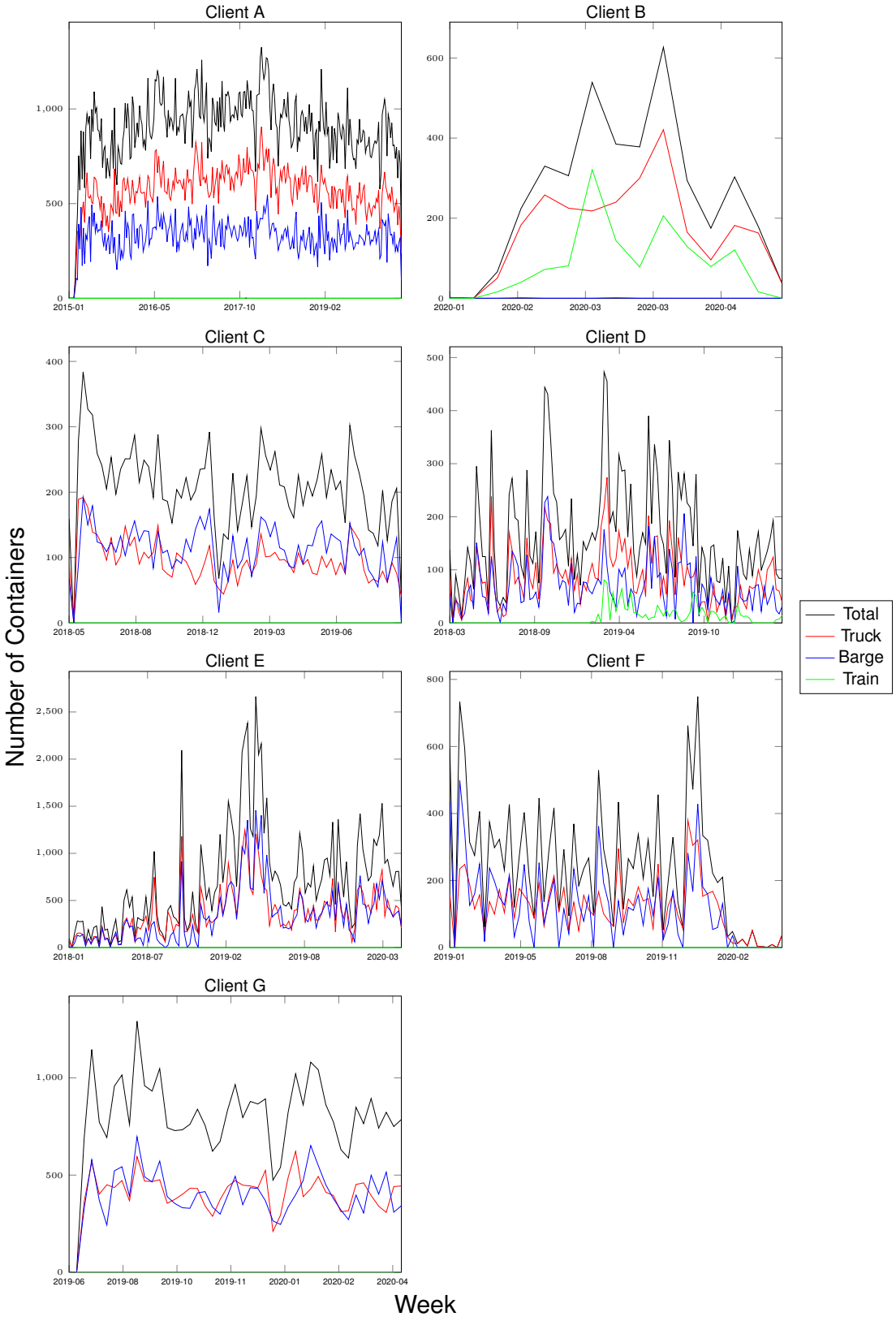

Figure 2: Overview of the number of containers handled per week. The colour indicates via which modality the containers departs.

A.3 Handled Containers per TEU

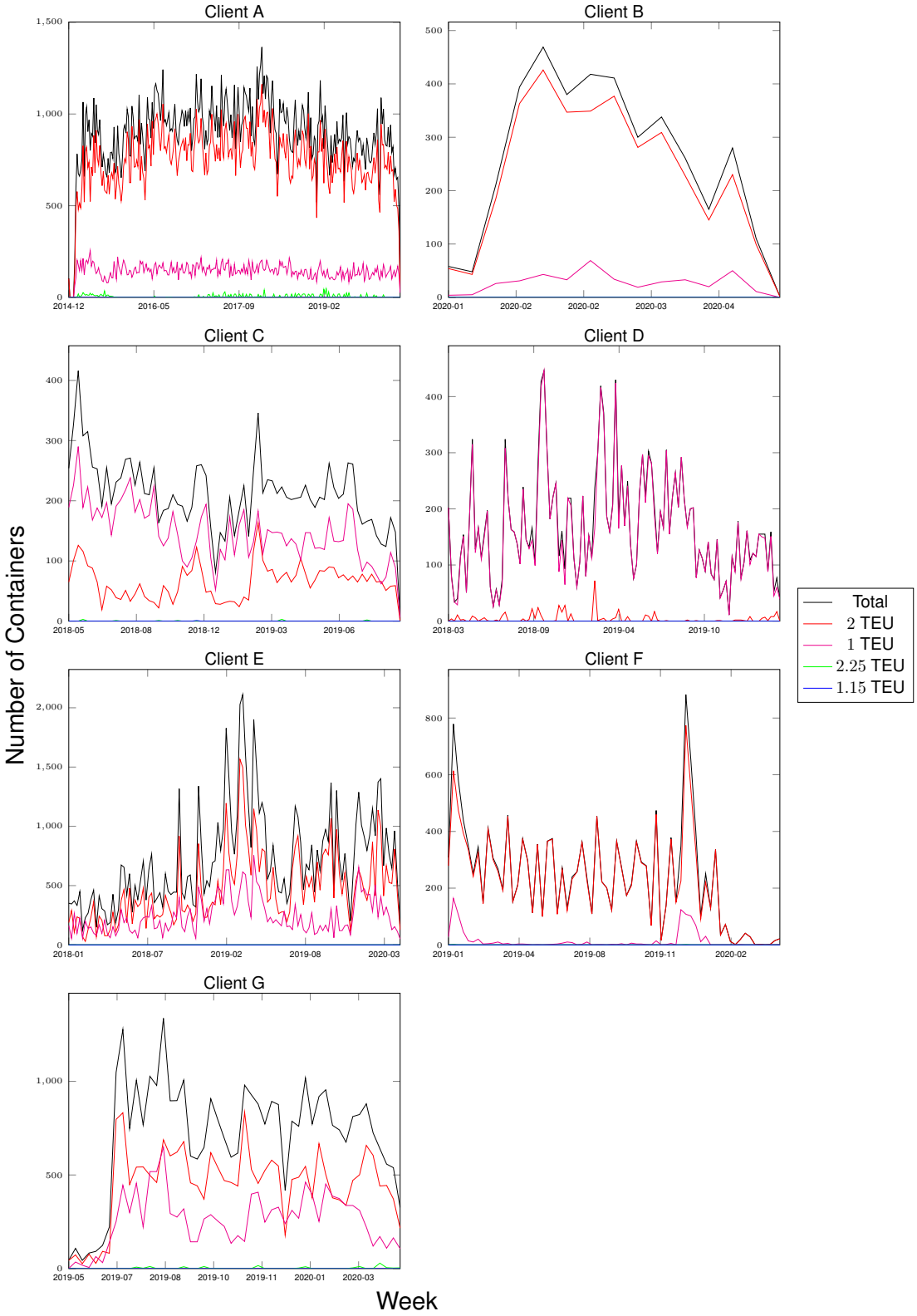

Figure 3: Overview of the number of containers handled per TEU per week.

A.4 Handled Containers per Category

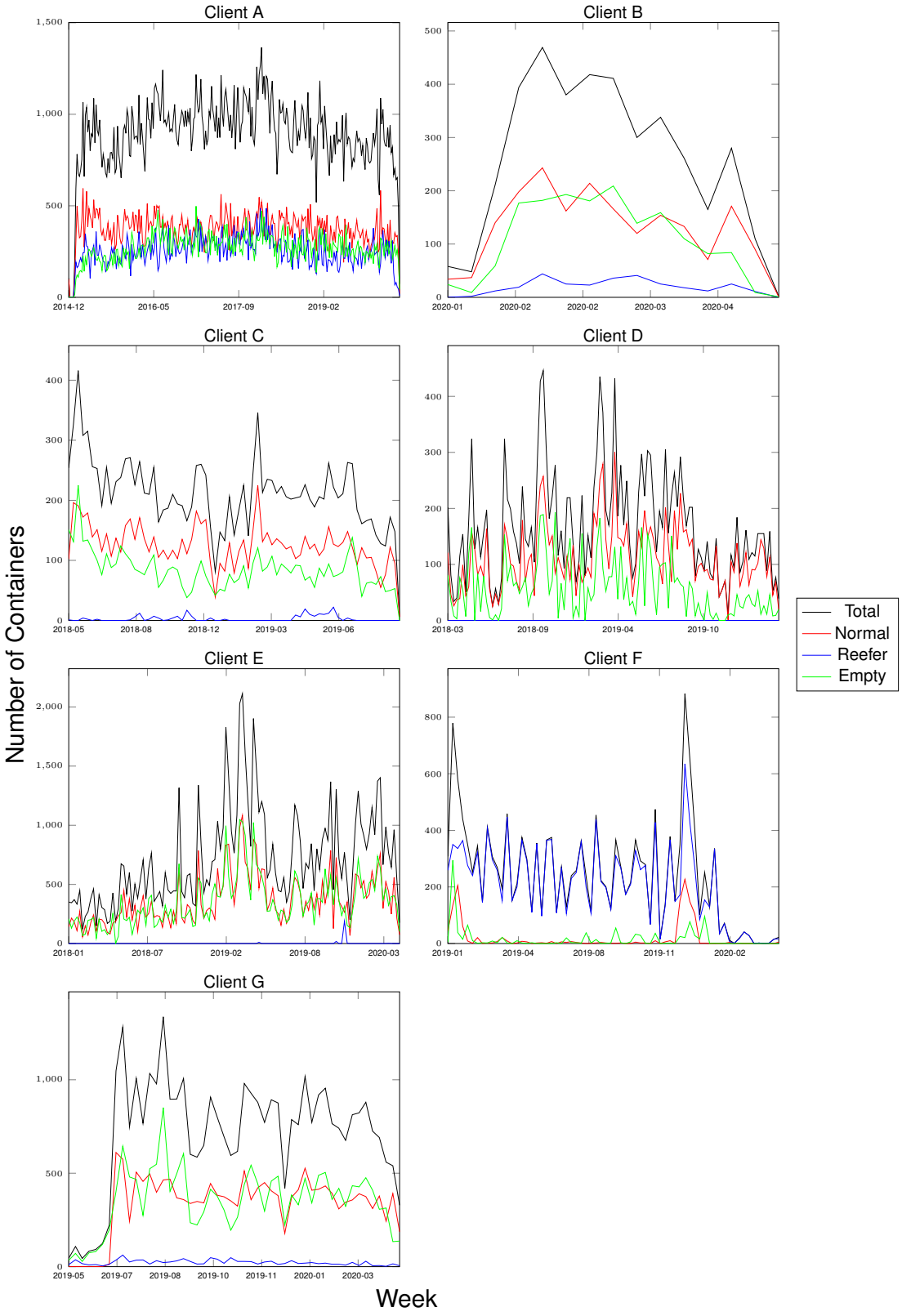

Figure 4: Overview of the number of containers handled per date per category.

A.5 Handled Reefers per TEU

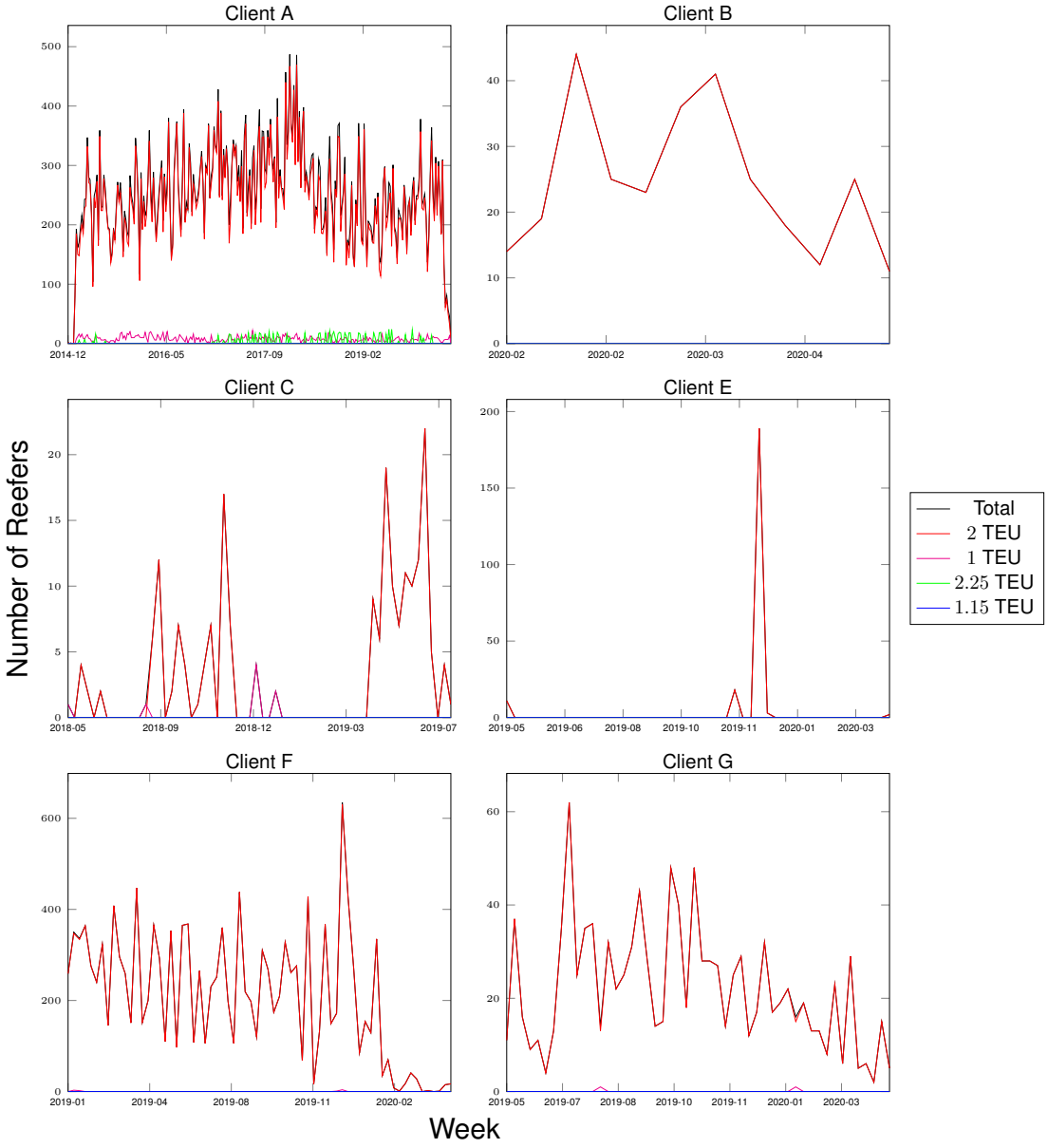

Figure 5: Overview of the number of reefers handled per TEU per week. Client D had no recorded reefer shipments and therefore not displayed.

A.6 Average Number of Containers handled per Weekday

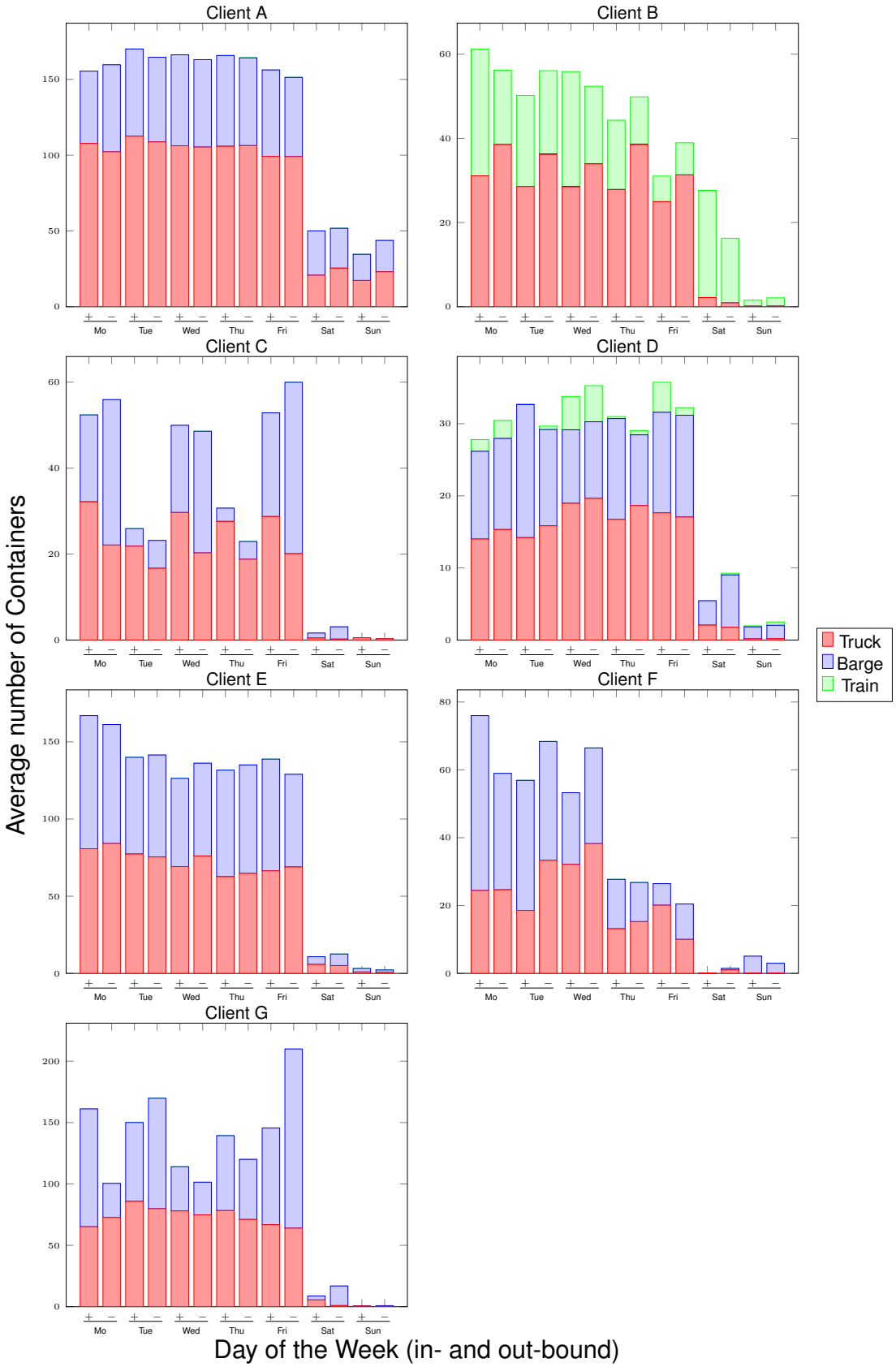

Figure 6: Overview of the average number of containers handled per weekday. A $+$ indicates the number of inbound containers handled and the $-$ the number of outbound containers handled.

A.7 △ Expected Arrival Time per Modality

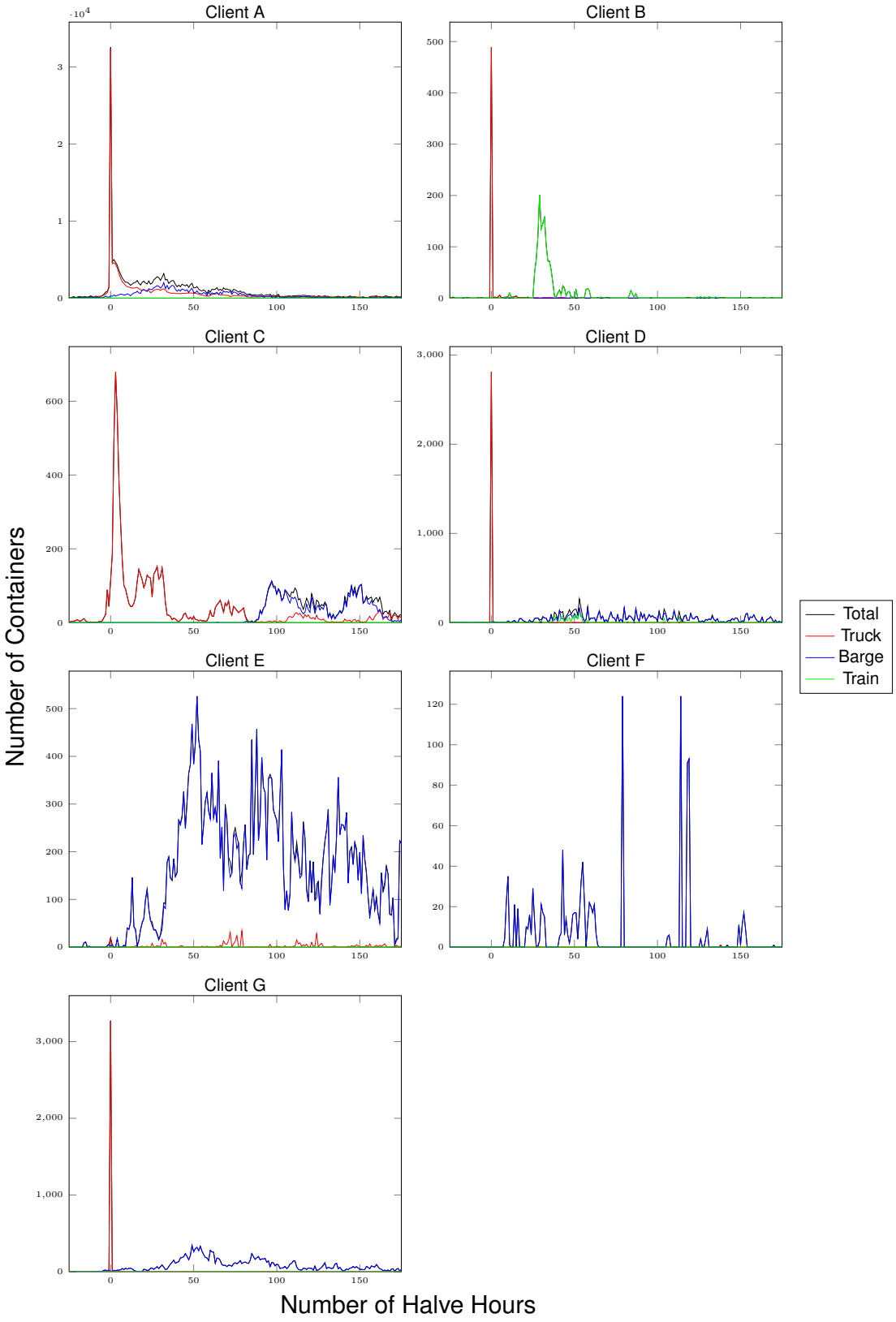

Figure 7: Overview of the number of containers that arrive later than the expected departure time per modality grouped per halve hour.

A.8 △ Expected Departure Time per Modality

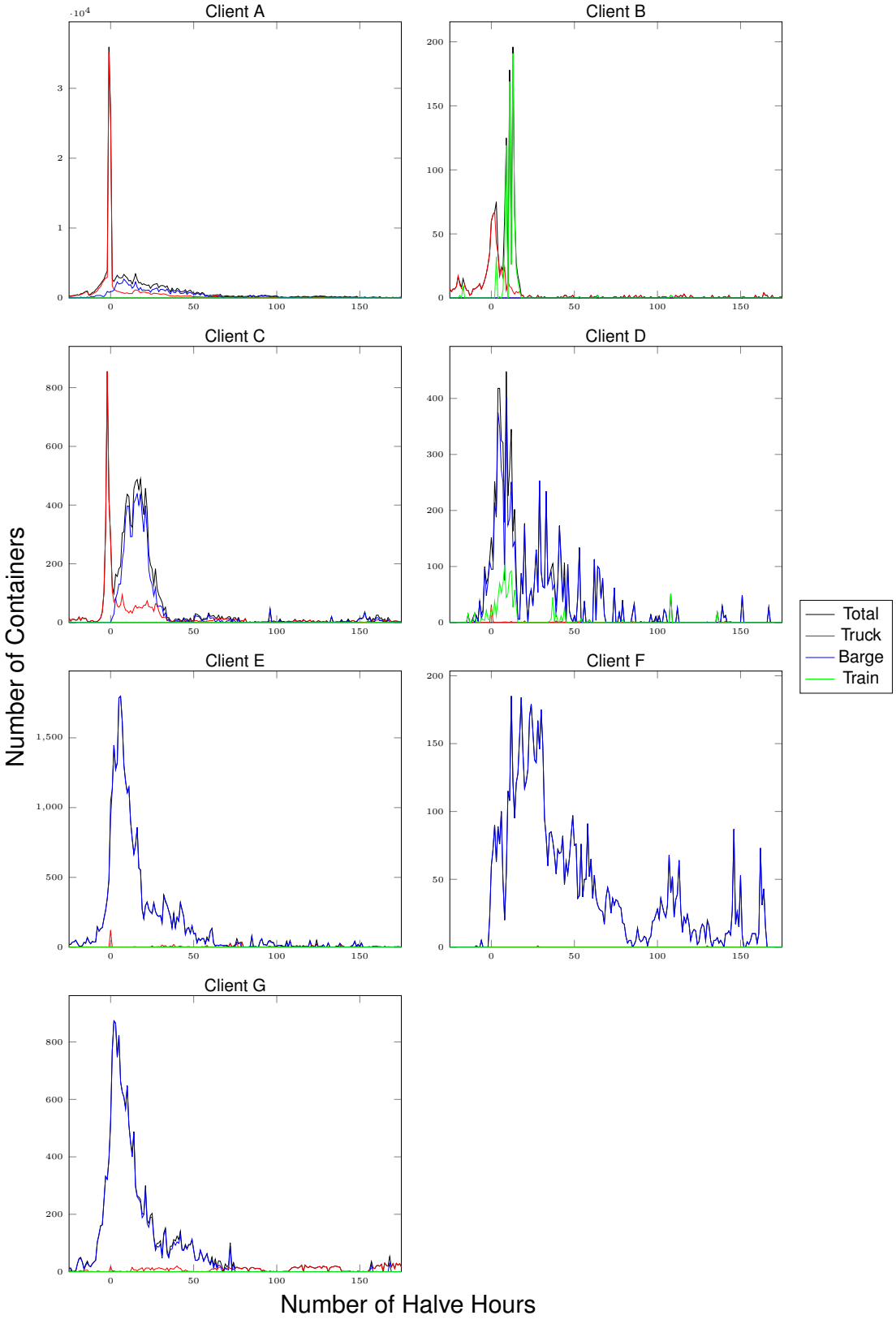

Figure 8: Overview of the number of containers that depart later than the expected departure time per modality grouped per halve hour.

A.9 Dwell Time per Container Type

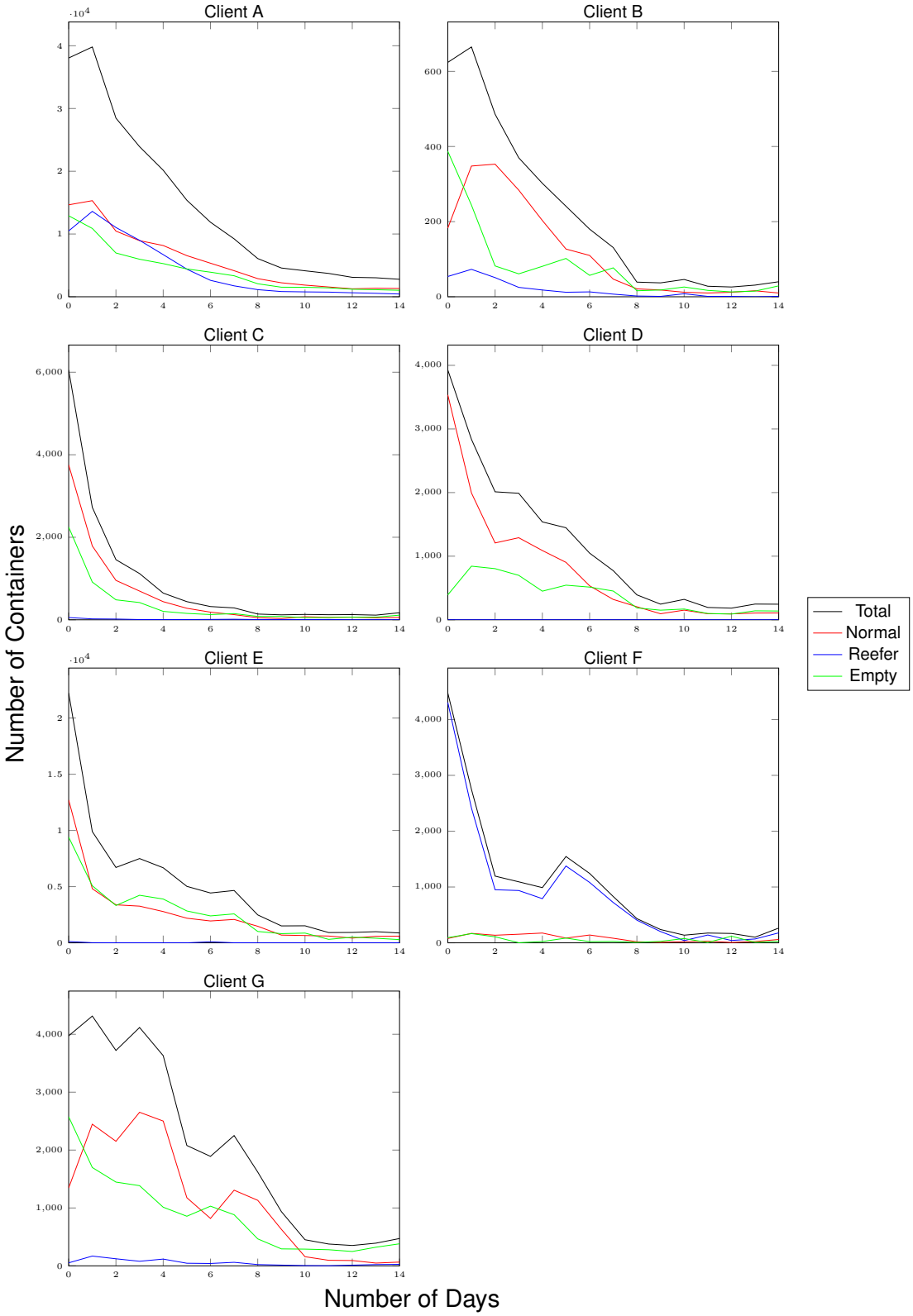

Figure 9: Overview of the number of days a container remains in the terminal.

A.10 Percentage of Mixed Stacks

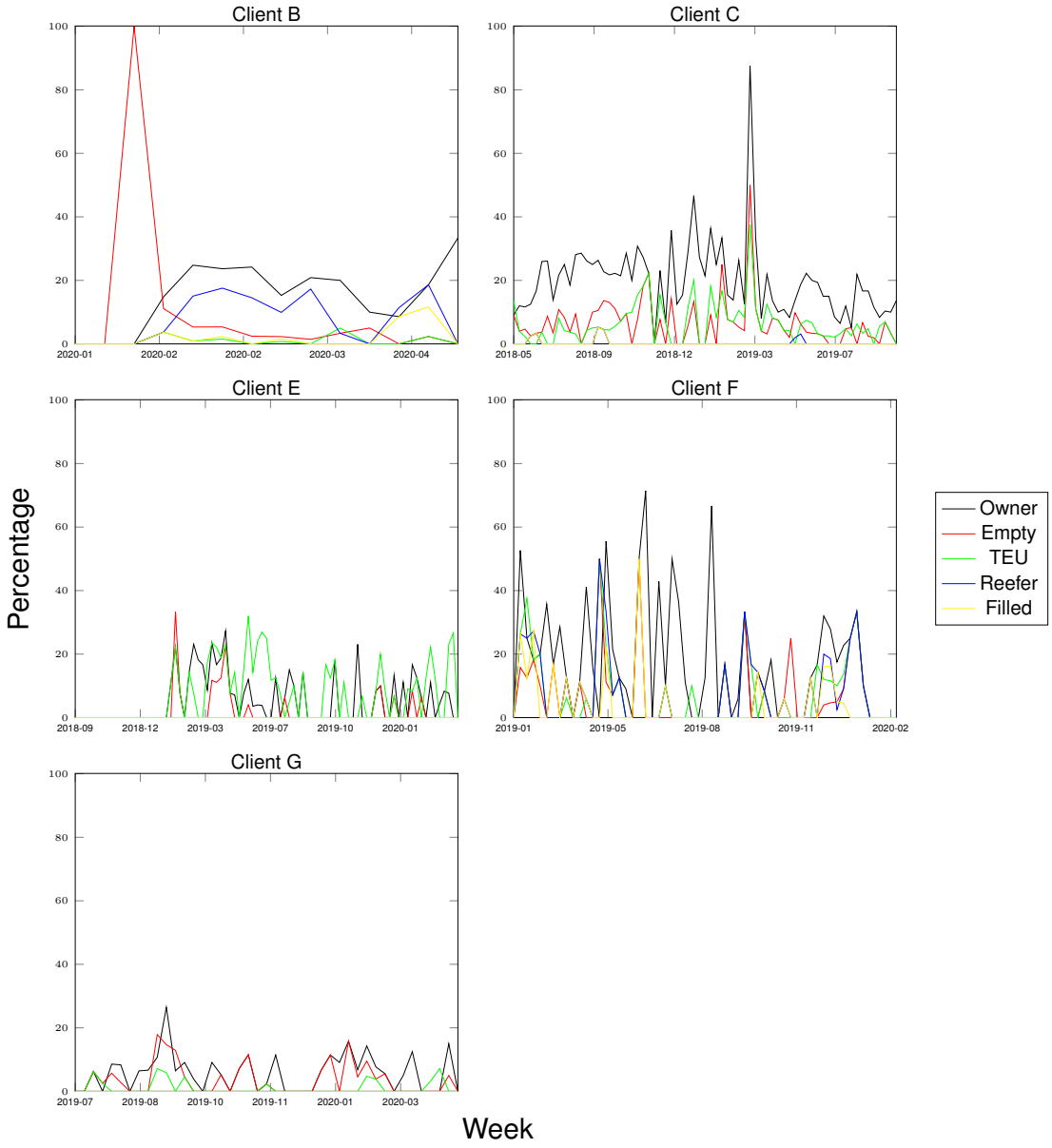

Figure 10: Overview of the percentage of mixed stacks according to the specified feature per week.

A.11 Number of Reported Reshuffles per Week

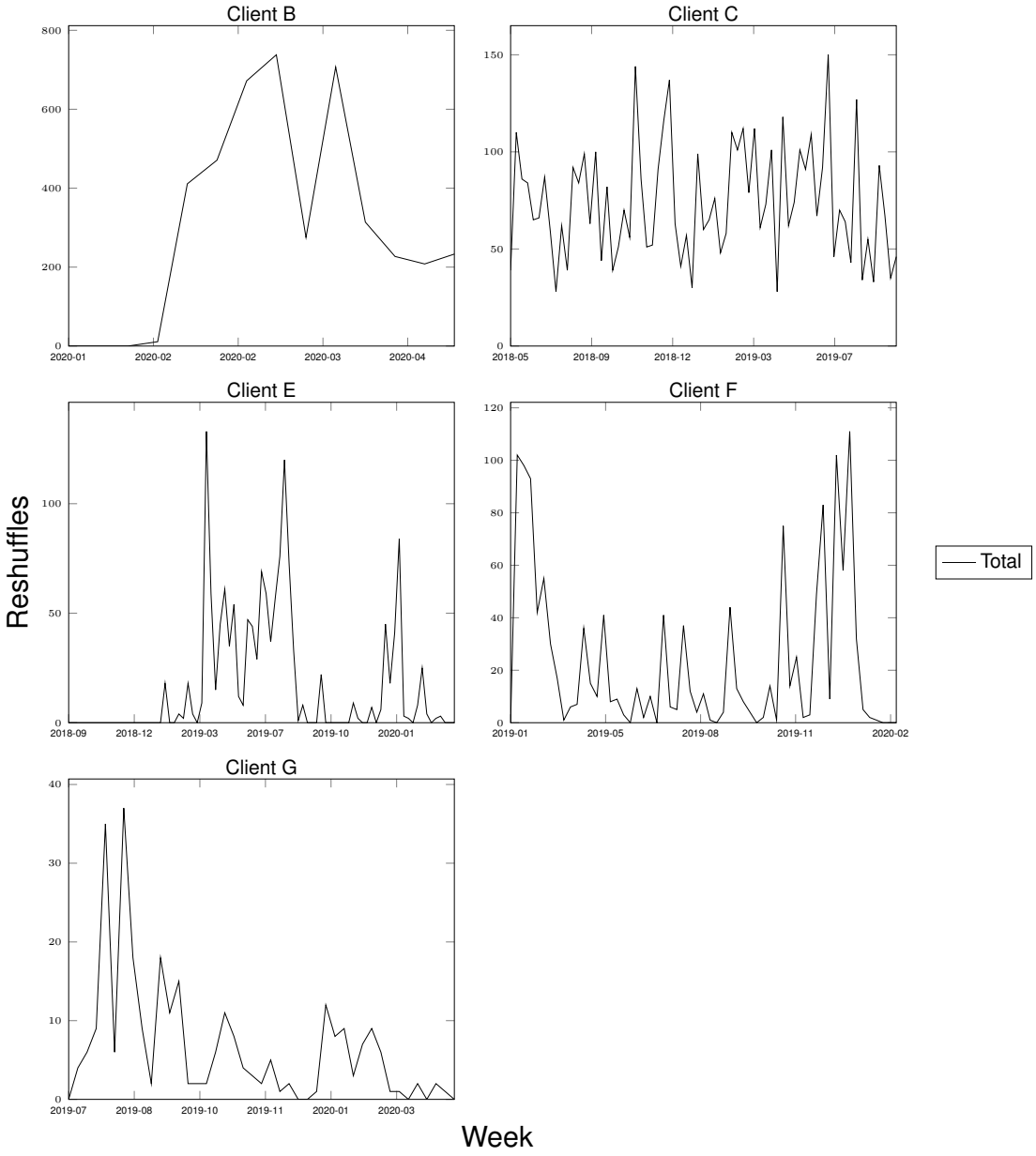

Figure 11: Overview of the number of reported reshuffles per week.

A.12 Number of Residing Containers

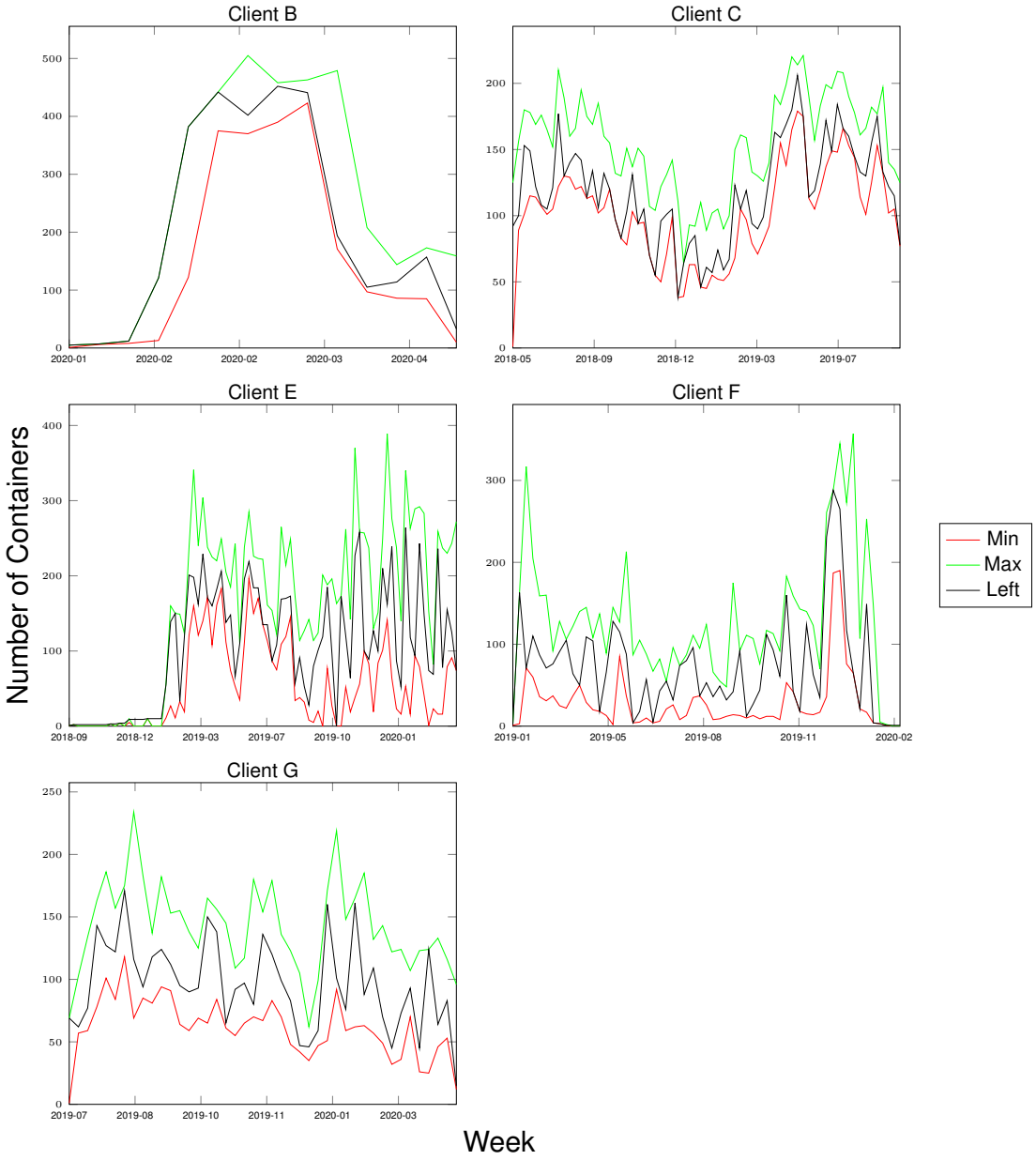

Figure 12: Overview of the number of containers that remain at the terminal per week. The min and max indicate respectively the minimum and the maximum number of containers that occupied the terminal that week. Left indicates the amount present at the end of the week.

A.13 Results Terminal Layout 1 & 2

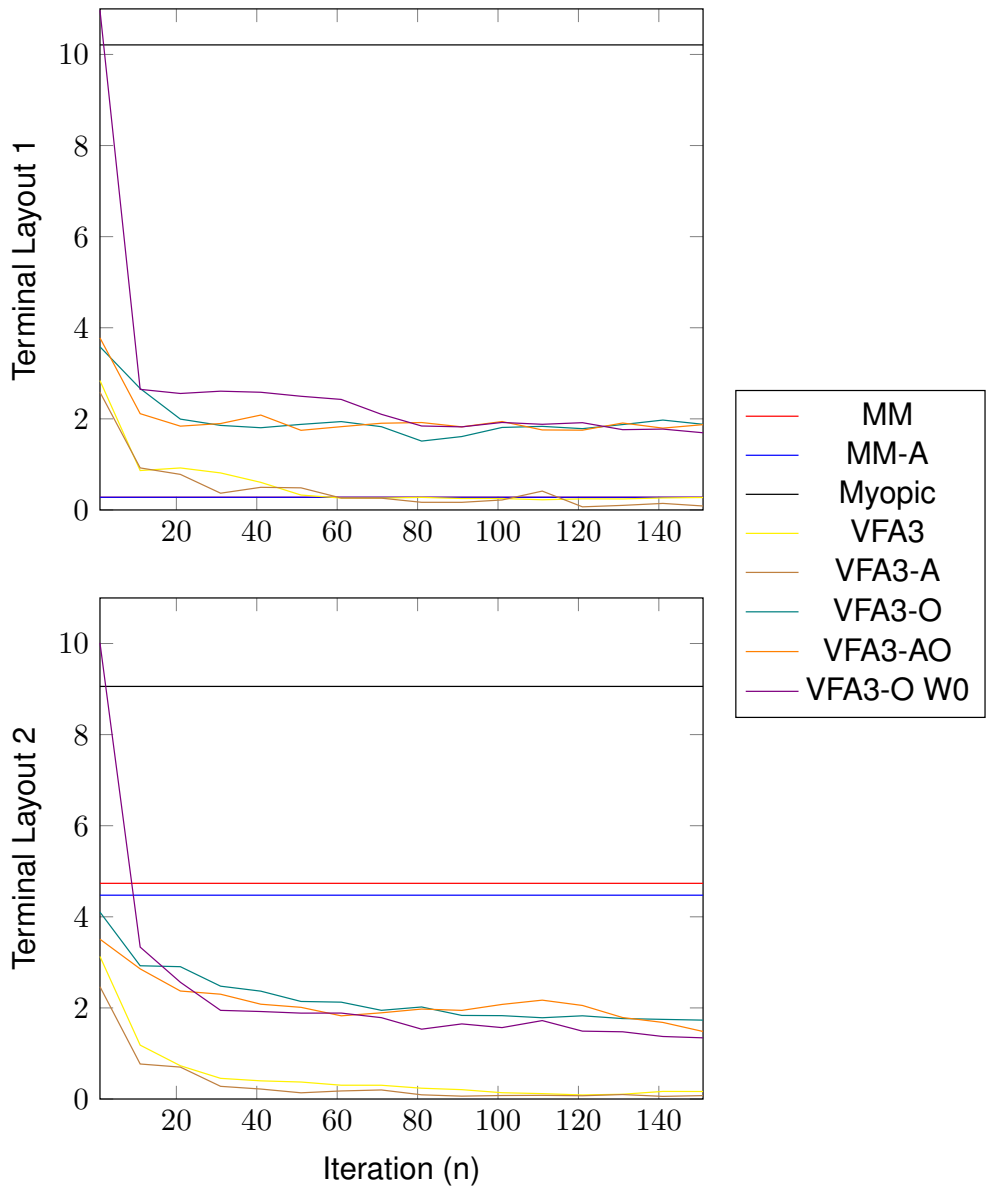

Figure 13: Graph showing the number of reshuffles of the various algorithms on the smaller problem instances. VFA3-O W0 represents VFA3-O with weights initialized at zero.