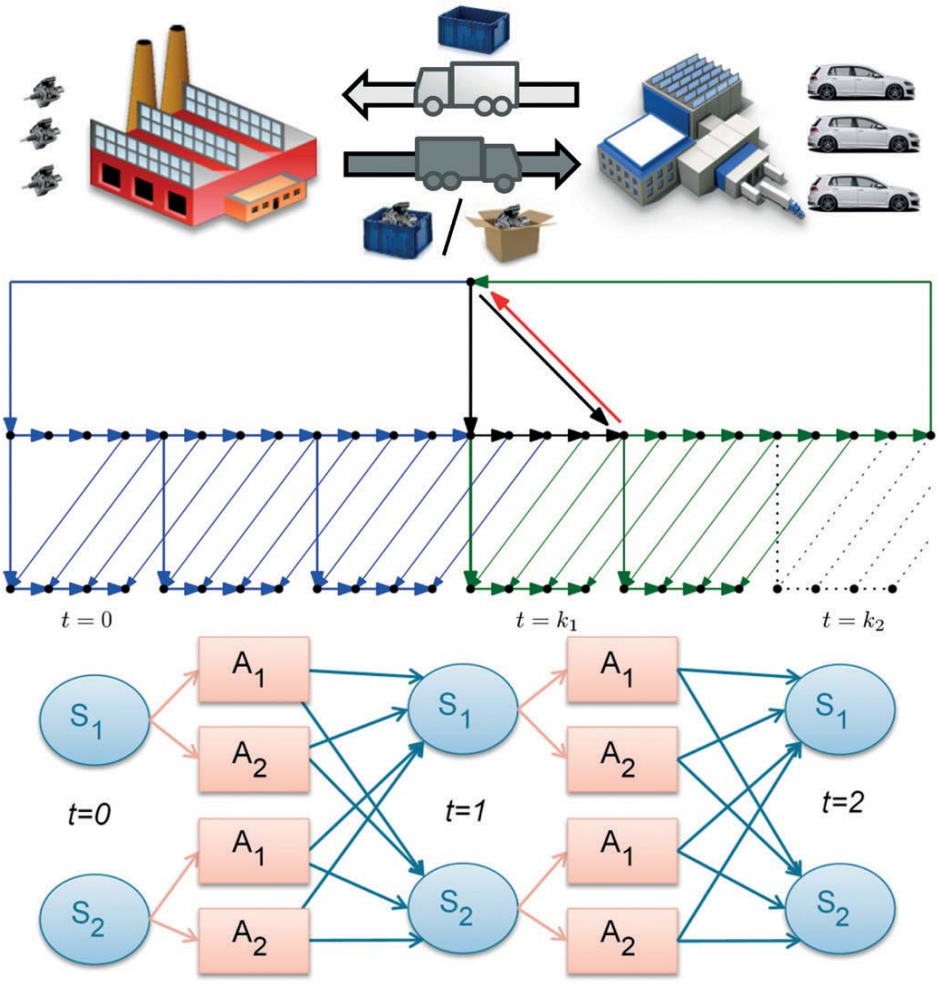
Neil Jami

Container Fleet Management in Closed-Loop Supply Chains



Technische Universität Kaiserslautern

Fraunhofer ITWM

Fraunhofer-Institut für
Techno- und Wirtschaftsmathematik ITWM

Container Fleet Management in Closed-Loop Supply Chains

Neil Jami

FRAUNHOFER VERLAG

Kontakt:
Fraunhofer-Institut für Techno- und Wirtschaftsmathematik ITWM
Fraunhofer-Platz 1
67663 Kaiserslautern
Telefon +49 631/31600-0
Fax +49 631/31600-1099
E-Mail info@itwm.fraunhofer.de
URL www.itwm.fraunhofer.de

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.
ISBN (Print): 978-3-8396-1210-1

D 386

Zugl.: Kaiserslautern, Univ., Diss., 2016

Druck: Mediendienstleistungen des
Fraunhofer-Informationszentrum Raum und Bau IRB, Stuttgart

Für den Druck des Buches wurde chlor- und säurefreies Papier verwendet.

Fraunhofer-Informationszentrum Raum und Bau IRB
Postfach 80 04 69, 70504 Stuttgart
Nobelstraße 12, 70569 Stuttgart
Telefon 07 11 9 70 - 25 00
Telefax 07 11 9 70 - 25 08
E-Mail verlag@fraunhofer.de
URL http://verlag.fraunhofer.de

Container Fleet Management in Closed-Loop Supply Chains

Vom Fachbereich Mathematik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften
(Doctor rerum naturalium, Dr. rer. nat.)
genehmigte

Dissertation

von

Neil Jami

1. Gutachter: Prof. Dr. Karl-Heinz Küfer
2. Gutachter: Prof. Dr. Jörg Rambau

Datum der Disputation: 3. März 2017

D 386

Acknowledgements

This work has been done with the financial support of the department Optimization of Fraunhofer Institute for Industrial Mathematics (ITWM).

I would like to express my sincere gratitude to my supervisor Prof. Dr. Karl-Heinz Küfer for giving me the opportunity to work on this PhD thesis in his department and for the very insightful comments on my research. I thank everyone form the optimization department for the great environment that really made me feel confortable in my research.

Special thanks to Dr. Michael Schröder, who introduced me to this challenging research topic and for the many fruitful discussions that helped me to define a research direction.

I am also grateful to Jasmin Kirchner for helping me through many administrative procedures, which allowed me to concentrate on my thesis.

Besides, I thank Martin Berger, Bastian Bludau, Esther Bonacker Elisabeth Finhold, Tino Fleuren, Michael Helmling, Anna Hoffmann, Helene Krieg, Dimitri Nowak, Rasmus Schroeder, and Phillip Süss for giving me advises and helping with some proof-reading.

Finally, I want to thank my family for their love and support.

Kaiserslautern, September 2016

Contents

Chapter 1

Introduction

The objective of this thesis is to develop models and algorithms to plan the purchasing of reusable containers in a closed-loop supply chain where the demand is increasing. This first chapter introduces the topic of container management and describes the contribution of this thesis.

1.1 Motivation

The work in this thesis is motivated by a collaborative research project in the automotive industry. Our goal is to improve the flow of returnable containers between an original equipment manufacturer and his supplier. In this closed-loop supply chain, the supplier provides some car parts to the manufacturer on a daily basis. For safety and quality reasons, these items are transported in containers stacked on pallets (see Figure 1.1).

Figure 1.1: Pallet containing several containers

The manufacturer owns the containers and keeps most of them in his inventory. He periodically provides empty containers to the supplier. In the agreement, the manufacturer sends empty containers back to the supplier on a weekly basis. Because it is expensive for the supplier to hold many containers, he should keep as few containers as possible. Consequently, there is a need to manage containers in the logistic chain and decide how many empty containers should be sent to the supplier.

Furthermore, the actors of the supply chain have noticed a rapid increase of the demand over time (see Figure 1.2). The second objective of this use case is to determine when the manufacturer should purchase new returnable containers, and how many containers to add to the system.

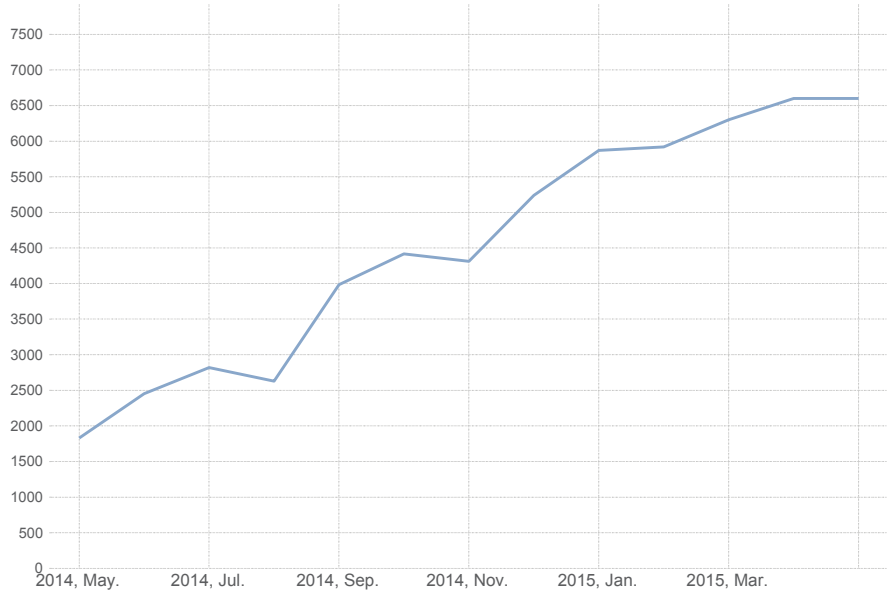


Figure 1.2: Evolution of the expected demand over time.

Shortage of items must be avoided, since stopping the production chain of cars would incur a huge cost to the manufacturer. Backlogging is not allowed and the supplier must ensure that his production is enough for the worst case scenario. This is a necessary cost in the supply chain. However, shortage of containers is possible, and the supplier has the possibility to buy one-way packages like cardboard boxes to transport the car parts. While this alternative is definitely profitable is some cases, the use of returnable containers is favored for safety, cost, and environmental reasons. Cardboard boxes are disposed by the manufacturer after their arrival and the manufacturer must take a special care of the one-way packages, which incurs an additional cost.

1.2 Literature Summary

Inventory management problems are one of the most studied scientific topics. We cite for example "The logic of logistics" from Simchi-Levi et al. [108], "Foundations of Inventory Management" of Zipkin [143] and "Principles of Inventory Management" from Muckstadt, [82] among many. Grünert and Irnich [34] describe the basic problems and models in transportation logistics as well as the main optimization tools to solve these problems. Some other books like "Approximate Dynamic Programming" from Powell [90] focus on a general algorithmic framework and consider applications in logistics, because logistic problems are very challenging and it requires a lot of efforts to obtain a good solution. This thesis builds up on two main streams of research: lot-sizing problems and empty container repositioning problems.

Lot-sizing problems consist in deciding how many items we should buy at each time period to face future demands, given a setup cost at purchasing and a linear holding cost. A fundamental characteristic of lot-sizing problems is that, because of the setup cost when purchasing new items, the optimal policies generally do not purchase items at every period. An important part of developing algorithms consists in deciding at which periods the purchasing should be done.
A fundamental work on lot-sizing problems is from Wagner and Within [128], in 1958. They introduce the dynamic lot-sizing problem, where the demand is deterministic and evolves over a finite time horizon. The authors develop a dynamic programming based algorithm, which we describe in this thesis and adapt to our use case. This algorithm is considered as one of the most influential papers in management science (see Hopp [47]), and many studies have extended it to more complex models.
When the demand is random, the problem is much more difficult to solve. Bookbinder and Tan [11] describe three possible strategies to solve the dynamic lot-sizing problem under stochastic demands.
In our use-case, an interesting extension of the dynamic lot-sizing problem includes remanufacturing. A fixed amount of items returns to the manufacturer. These returned items can be converted back into 'serviceable items' and be sold again. Nonetheless, lot-sizing problems with remanufacturing assume no relationship between the number of sent items and the number of returning items, which is a fundamental difference with our problem. In our case, the manufacturer only needs to purchase new containers corresponding to the demand increase. Kiesmüller and Van der Laan [63] present an inventory control problem where sold products return with some probability to be then remanufactured and sold again. They show that neglecting the relationship between demands and returns frequently leads to bad performances.

On the other hand, there is also a lot of literature on empty container

repositioning problems for both deterministic and stochastic demands. We refer to Cimino et al. [17] for a state of the art on this topic. The paper of Kroon and Vrijens [65] sensitizes about the importance of returnable containers in supply chains and introduces several models to use them.
In the deterministic setting, the problem is usually formulated either as an integer linear program or as a network flow. In the stochastic setting, many papers write either a Markov decision process or a stochastic mathematical program. Since the stochastic problem is at least NP-hard, an approximation is frequently computed either using approximate dynamic programming (see Powell [90]), or with the help or a meta-heuristic like the gradient method (see Dong and Song [22]) or the sample average approximation (see Kleywegt et al. [64]).

However, despite the close relationship between lot-sizing problems and empty container repositioning problems, very few studies consider the questioning of purchasing containers. From our point of view, this lack of research comes from two reasons:

1. Firstly, most container management problems have a huge state space, and are very complex to solve due to the resulting curse of dimensionality (see Powell [90]). Consequently, many researchers rather consider a short time horizon. They can then suppose that every container is purchased right from the start, without deteriorating much the solution.

2. Secondly, most of the research on empty container repositioning is between maritime ports, where the number of containers is known to be largely sufficient for a middle term planning. The main issue in maritime container management is the imbalance of the exchanges between regions. Thus, there is an important need to send empty containers back to heavy exporting regions rather than purchasing new containers.

Rather than considering a ramp-up scenario with a growing demand, research papers either consider a stationary demand or add an option of leasing containers from a third party provider. Alternatively, some studies assume that the container fleet size is unlimited and that containers can be stored into depots at a low holding cost. There is a branch of literature called *container fleet sizing*, where the objective is to optimize the number of containers in a generally stationary system.

1.3 Goals of the Thesis

The main goal of this thesis is to propose a first attempt at optimizing the purchasing plan of returnable containers. We consider both cases of

a deterministic demand and of a stochastic demand. Our work is mostly restricted to a logistic chain between a single manufacturer and a single supplier. We assume that there is a central decision maker in the supply chain aiming at minimizing the joint cost of the two partners. The total cost is then possibly redistributed between the partners, but that is not the subject of the thesis.
We put a strong accent on two criteria. Firstly, because of the growing demand on the market, we would like to avoid purchasing every container at the start. Instead, we gradually increase the container fleet size. Nonetheless, purchasing new containers too frequently disturbs the stability of the supply chain and brings more administrative work. Because of that, we consider a high setup cost for each container purchasing to models administrative and management costs. Secondly, we consider positive lead times. We give a lot of importance to this assumption, because in our application research project, it takes around two days to transport the items to the manufacturer, whereas the empty containers are sent every week. Consequently, the number of outgoing full containers should have a strong impact on the optimal container fleet size. In this thesis, we show that the problem is much easier to solve without these assumptions.

1.4 Outline of the Thesis

This thesis is divided into two introductory chapters and three parts, as illustrated in Figure 1.3. Chapter 2 provides a mathematical description of our container management problem and explains some properties making this problem challenging to solve. Moreover, we explain the dynamic programming framework used in the Wagner-Within algorithm which we use all along this thesis.
Our first contribution regroups Chapters 3 to 6 and solves the problem when the demand is deterministic. Every study in this part is based on flow networks. The deterministic part presents and extends the results from one of our papers [54]. More precisely, Chapter 3 provides a literature overview, introduces the reader to the network flow theory, and formulates the problem as a minimum cost flow on a network with fixed-plus-linear cost. However, there is no efficient algorithm in the literature to solve this type of flow problem. Chapter 4 presents a first resolution of the problem, where we assume that the demand is steadily increasing. Under the hypothesis that no disposable is bought for shortly after purchasing new containers, we solve the problem in polynomial time. Algorithm $Flow.4.1$ is the central result of our paper [54] and runs in $O(T^4 \cdot R^2 \cdot \log[R \cdot T]^2)$ time. This algorithm is similar to the Wagner-Within algorithm but computes minimum linear-cost flows at each iteration. We extend it to different settings such as multiple suppliers and a bi-objective function. We adapt it to compute

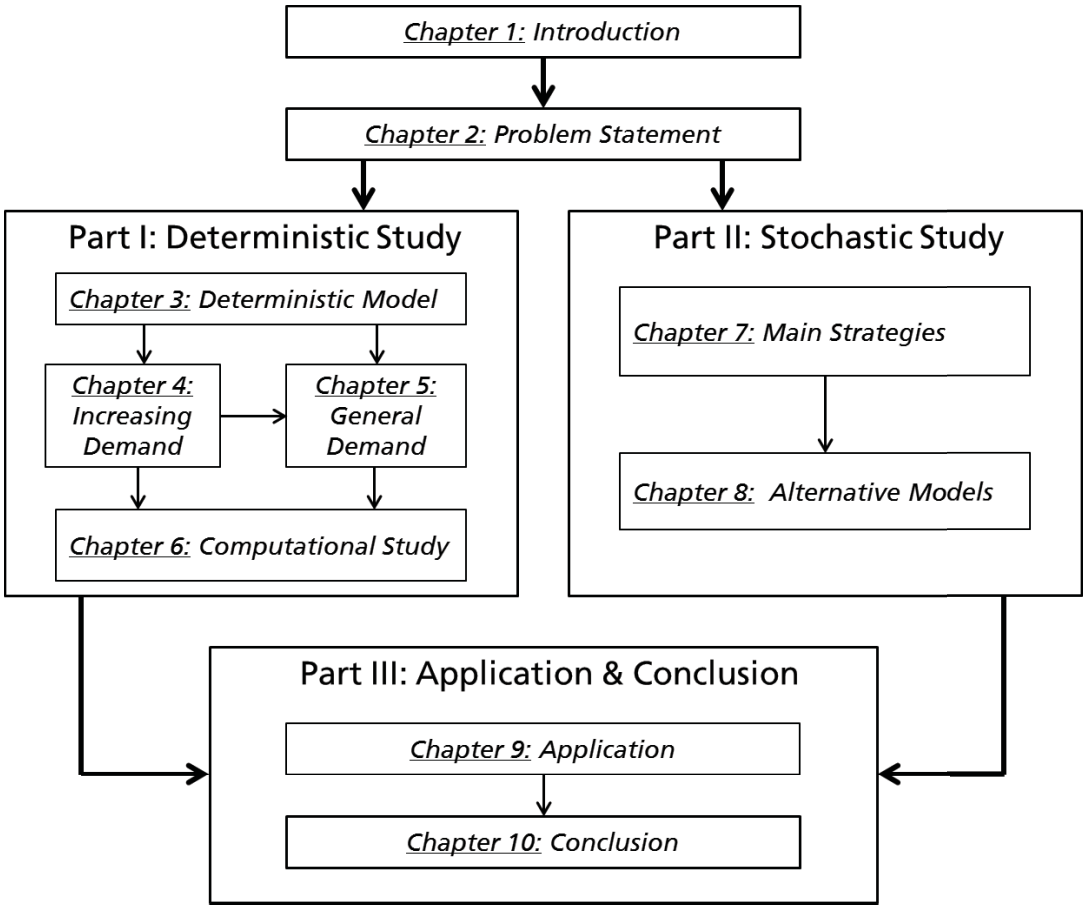


Figure 1.3: Outline of the thesis

a feasible solution when the demand is not always increasing, but at the cost of the optimality property. Chapter 5 is dedicated to a general demand pattern. We develop alternative algorithms to those from Chapter 4. These algorithms are optimal under the hypothesis that at least one container must be idle at each purchasing period. We also propose a very general framework to solve the problem. In Chapter 6, we finally present a computational study highlighting the efficiency of our algorithms. In addition, we propose three ways of improving the running time of our algorithms. We consider parallel computing, an more compact network representation, and a new algorithm to compute a minimum linear-cost flow in our flow networks. This algorithm uses the cycle-canceling framework and is a notable result of this chapter, as it outperforms the best algorithm of the literature when applied to our model.

The second part of this thesis includes Chapters 7 and 8. It analyzes the problem under a stochastic demand. This contribution is based on a second paper [55]. Chapter 7 develops a literature review in this setting and proves that the problem is NP-hard. We extend the three lot-sizing strategies described by Bookbinder et al. [11] to our container management context. We

complete our solution framework with a fourth strategy corresponding to the Silver-Meal heuristic (see Silver and Meal [107], Silver [106]). Our algorithms use a Markov decision process framework. The online algorithm runs in $O(T \cdot R^5 \cdot D^6_{\max})$ time and the other algorithms in $O(T^2 \cdot R^4 \cdot D^4_{\max})$ time. In Chapter 8, we discuss alternative algorithms and ideas from the literature, to compute a faster solution. This chapter opens up several research directions which are not developed during my thesis.
The final third part provides an outlook on the thesis. Our last contribution in Chapter 9 is the application of different policies from this thesis to a real-life scenario in the automotive industry. Chapter 10 concludes the thesis, highlighting its contributions and future research directions.

Chapter 2

Problem Description

This chapter formally defines our container management problem, which we call container purchasing problem (CPP). Three assumptions make the problem more realistic but also more complex. Firstly, the supplier does not need to exactly match the demand in containers as he can buy single-use disposables. Secondly, the transportation of containers between the two actors takes some time, which requires a more careful use of containers. Thirdly, container purchasing comes at a setup cost, so the manufacturer does not want to purchase new containers at every period. Afterward we describe the Wagner-Within Algorithm (W-W). Finally, we explain some notations we use for our simulations.

2.1 Problem Statement

We consider a cooperation in a closed-loop supply chain where items have to be transported in *packages*. Every package has the same size and the same type, so any package can be used to transport any item. A package can be either in a returnable container, which we call *container*, or a single-use disposable container like a cardboard box, which we call *disposable*.
We focus on a supply chain between a single supplier and a single manufacturer. The supply chain is described in Figure 2.1. The manufacturer can purchase new containers at a setup cost, while disposables are directly available at the supplier location. For clarity, we use the verb *purchase* exclusively for containers and the verb *buy* only for disposables.
The time horizon is finite and composed of T periods of R time steps each. The periods are indexed $t \in \{0, \dots, T-1\}$ while the steps are indexed $r \in \{0, \dots, R-1\}$. Since every variable is discrete, we use the traditional interval notations for integral numbers. The *time* (t, r) corresponds to step $r \in [0, R[$ of period $t \in [0, T[$, which is the $(R \cdot t + r)$-th point of the time horizon. We extend the definition of the time to $r \in \mathbb{Z}$ so that time (t, r) equals time $(t + 1, r - R)$. In the thesis, we frequently use the notation

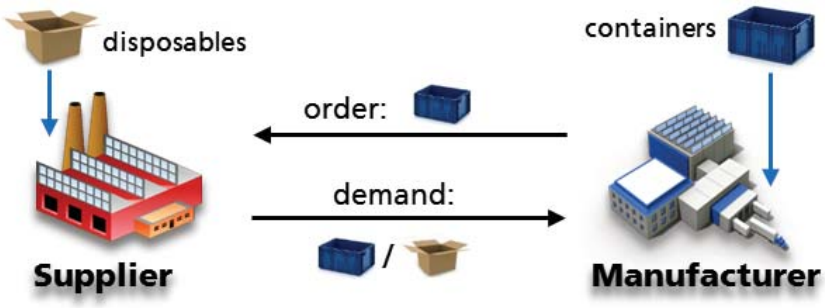


Figure 2.1: Closed-loop supply chain

$(t, r+r')$ with $r \in [0, R[$ to denote the point in time r' steps after time (t, r), even if $r+r' \geq R$. We use the wordings *up to* and *from* period t (respectively time (t, r)) when we include period t (respectively) time (t, r)), and the words *before* and *after* when we exclude it. For instance, the expression "from time t_1 to time t_2"denotes the time interval $[(t_1, 0), (t_2, R-1)]$, whereas "before period t"represents the periods $t' < t$.
We denote by *demands* the flow of items in packages from the supplier to the manufacturer, and by *order* the flow of empty containers from the manufacturer back to the supplier. A demand $D_{t,r}$ (or $D(t, r)$) occurs during every step $r \in [0, R[$ of every period $t \in [0, T[$. We use the notation $D(t, r)$ when the demand is deterministic and $D_{t,r}$ for stochastic demand. We denote respectively by α_t and β_t the order size and the purchase size at period $t \in [0, T[$. At time step $r = 0$, the purchasing occurs first, followed by the ordering and finally the demand. We extend the definition of $D(t, r)$, α_t and β_t to $t \in \mathbb{Z}$ so that:

$$\forall t \notin [0, T[,\ \forall r \in [0, R[:\ \alpha_t = \beta_t = D(t, r) = 0 \tag{2.1}$$

We decompose every demand $D(t, r)$ into *demand units* $D(t, r, i)$ with $i \in \mathbb{N}$ so that each demand unit requires the use of one package. We use the term *satisfying* a demand as a general term meaning that we send the corresponding number of items in packages of any type. In contrast, we say that a demand unit is *fulfilled* if the item is put in a container and not in a disposable. We say that a container *fulfills* a demand unit if it is used to transport the item. Without loss of generality, we always fulfill $D(t, r, i)$ before $D(t, r, i+1)$.
We denote by *ordering delay* $L_{ord} \geq 0$ the (integral) number of steps needed to send empty containers from the manufacturer to the supplier, and by *delivery delay* $L_{del} \geq 1$ the (integral) number of steps to sent full containers to the manufacturer. In our model, the ordering is done at the beginning of each time step (t, r) We suppose that the decisions α_t and β_t are taken exactly at the beginning of time $(t, 0)$ whereas the demands $D(t, r)$ are satisfied during time (t, r), i.e. between time (t, r) and time $(t, r+1)$. Consequently,

for an identical ordering and delivery delay in practice, the full containers relative to demand $D(t, r)$ leave the supplier after time (t, r) and arrive after time $(t, r + L_{del})$, so these containers can only be ordered starting from time $(t, r + L_{del} + 1)$. On the other hand, the empty containers leaving the manufacturer at time $(t, 0)$ arrive at time (t, L_{ord}), so before demand $D(t, L_{ord})$ is satisfied. Thus, for the same transportation time in both directions, we will have: $L_{del} = L_{ord} + 1$. We assume that the ordering and delivery delays also include the time of every operation from the departure of the containers from an actor's stock to their arrival to the other actor's stock, like the loading, unloading and cleaning of containers.
We denote by *rotation* the cycle of a container in the process, from being ordered to its arrival back to the manufacturer as a full container. We assume that the minimum rotation time is lower than the ordering frequency:

Hypothesis 2.1 ***[Delay]:***

$$1 \leq L_{ord} + L_{del} \leq R \tag{2.2}$$

We refer to a container as *idle* during a period t if it is in the manufacturer stock at time $(t, 0)$ but is not ordered. This container hence stays in the manufacturer stock up to time $(t + 1, 0)$. A container which is not idle is said to be *busy*. A busy container may be transported, in the supplier stock or even in the manufacturer stock at time (t, r) if it arrived after time $(t, 0)$. When we decide that a container will be ordered, it will be busy for an integral number of periods.
At each time (t, r), the events occur in the following sequence:

1. The manufacturer purchases α_t containers, if $r = 0$.
2. The supplier orders β_t containers, if $r = 0$.
3. Empty containers arrive to the supplier, if $r = L_{ord} \geq 0$. In particular, the empty containers immediately arrive if $L_{ord} = 0$.
4. Full containers sent $L_{del} \geq 1$ steps ago arrive to the manufacturer.
5. The supplier buys disposables. The coming demand $D(t, r)$ is known before buying disposables, so the supplier knows exactly how many disposables he needs.
6. The demand $D(t, r)$ of time (t, r) is entirely satisfied, using disposables and containers.

We put no limit to the number of containers we can purchase and to the number of disposables we can buy. Moreover, we assume that a disposable cannot be kept in the supplier inventory, so that every disposable bought at time (t, r) must be immediately filled and sent to the manufacturer.

We denote by *container fleet size* u_t the number of containers in the system at period t before purchasing, and by u_t^α the container fleet size after purchasing. We define a *policy* as a function giving at each time step: the purchase quantity, the order size and the number of disposables bought by the supplier.
Our objective is to find a policy of purchasing and ordering returnable containers in the supply chain at minimum cost, when the demand is globally increasing. We say that a policy is *optimal* if it minimizes the joint cost of manufacturer and supplier.

2.2 Cost Structure

2.2.1 Notations

The costs of the system are separated into container purchasing costs, container holding costs and disposable costs. Buying a disposable at time (t, r) costs $C_{dis}(t, r)$. Container purchasing at the beginning of period t incurs a setup cost $C_{setup}(t)$ plus a price $C_{cont}(t)$ per container. We denote by *placement* a period where a positive number of containers is purchased. The holding cost of a container at the end of time (t, r) is $C_{sup}(t, r)$ for the supplier and $C_{man}(t, r)$ for the manufacturer. We suppose that containers are ordered without any cost. We can generalize our model to a constant cost $C_{ord} \geq 0$ per ordered container, as it is equivalent to decreasing the disposable cost by C_{ord}. The cost $C_{idle}(t)$ of a container being idle during period t is such that:

$$\forall t : \ C_{idle}(t) := \sum_{r=0}^{R-1} C_{man}(t, r) \tag{2.3}$$

We suppose that the manufacturer holding cost is positive, as otherwise we only need to purchase containers at the first period. We consider the following cost structure:

Hypothesis 2.2 ***[Cost]:*** *The manufacturer holding cost is lower than the supplier holding cost so that we keep the excess of containers at the manufacturer:*

$$\forall t, r, t', r' : \ 0 < C_{man}(t, r) < C_{sup}(t', r') \tag{2.4}$$

Moreover, the disposable costs are close enough to each other so that it is never optimal for the supplier to buy a disposable while holding a container (see Figure 2.2):

$$\forall t, r : \ C_{dis}(t, r) + C_{sup}(t, r) < C_{dis}(t, r + 1) + C_{man}(t, r + L_{del}) \tag{2.5}$$

The first assumption is called the *non-speculative cost structure*, which is generally assumed in the literature (see Chandoul et al. [12, 13]). It leads to the following result:

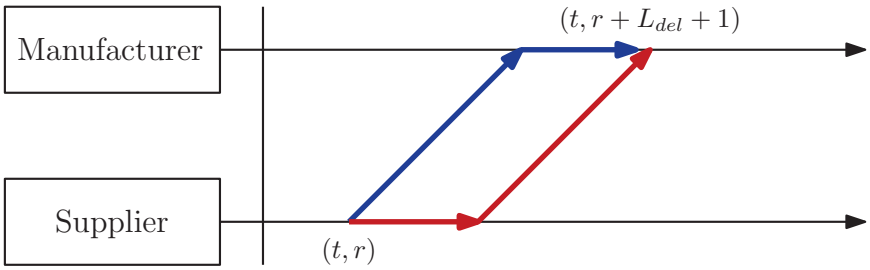


Figure 2.2: Illustration of the second cost assumption: it is more expensive for a container to follow the red path than the blue path.

Lemma 2.2.1 *Under Hypothesis 2.2 [Cost], it is always cheaper to order containers as late as possible:*

$$\sum_{r=0}^{R-1} C_{man}(t,r) + C_{ord}(t+1) \leq C_{ord}(t) + \sum_{r=L_{ord}}^{R+L_{ord}-1} C_{sup}(t,r) \tag{2.6}$$

Proof:

Suppose that a container is ordered at period t but only fulfills a demand at time $(t+1, L_{ord})$ or later. The total holding cost of the container from time $(t, 0)$ to $(t+1, L_{ord})$ is equal to the total supplier holding cost from time (t, L_{ord}) to $(t+1, L_{ord})$. By Hypothesis 2.2 [Cost], This cost is greater than the total manufacturer cost from time $(t, 0)$ to $(t+1, 0)$ which we would have paid if we ordered the container at time $(t+1, 0)$ instead.

□

We call the cost structure *stationary* if the cost functions C_{man}, C_{sup} and C_{dis} are constants. The second assumption of Hypothesis 2.2 is a consequence of the first one under stationary costs. When the cost functions are time dependent, we add this assumption to ensure that the supplier must favor the use of containers. In Figure 2.2, it is cheaper to send a container at time (t, r) and let it in the manufacturer stock for one step than keeping it to the supplier stock for one step before sending it.

2.2.2 Demand Profitability

We define the *profitability* $\mathbb{G}(t, r)$ of a demand $D(t, r)$ as the cost difference of satisfying one of its demand units between using a disposable and a container, assuming that a container is in the manufacturer stock. If we use a container, the cost of this container is equal to the supplier holding cost from its arrival to the supplier stock to its departure at time (t, r), plus the manufacturer holding cost from its arrival to the manufacturer stock to the next ordering time. If we use a disposable, the cost of using a disposable is the disposable price $C_{dis}(t, r)$ plus the cost of holding a container in the

manufacturer stock during the whole rotation. We have then:

$$\forall r \in [0, L_{ord}[:\ \mathbb{G}(t,r) := C_{dis}(t,r) + \sum_{r'=0}^{R+r+L_{del}-1} C_{man}(t-1,r') \tag{2.7}$$

$$- \sum_{r'=L_{ord}}^{R+r-1} C_{sup}(t-1,r') \tag{2.8}$$

$$\forall r \in [L_{ord}, R[:\ \mathbb{G}(t,r) := C_{dis}(t,r) + \sum_{r'=0}^{r+L_{del}-1} C_{man}(t,r') \tag{2.9}$$

$$- \sum_{r'=L_{ord}}^{r-1} C_{sup}(t,r') \tag{2.10}$$

We say that a demand is *profitable* if its profitability is positive. If a demand is not profitable, then it is always cheaper to satisfy it using exclusively disposables, no matter how large the container fleet size is. Therefore, we can assume without loss of generality that every demand is profitable.

2.3 Problem Properties

2.3.1 Effect of Positive Lead Times

Positive lead times add complexity to our container management problem. The issue is that some demands allow a faster rotation of the containers than others. If $r \in [L_{ord}, R - L_{del}]$, the rotation of the container finishes after one period. On the other hand, for $r \in [0, L_{ord}[\cup]R - L_{del}, R[$, the containers need one more period to be back to the manufacturer and ready to be ordered. We call a time step r and its corresponding demands $D(t,r)$ *preliminary* if $r \in [0, L_{ord}[$, *early* if $r \in [L_{ord}, R - L_{del}]$, and *late* if $r \in]R - L_{del}, R[$, as illustrated in Figure 2.3.

The rotation in the system takes two periods for preliminary and late demand while it only takes one period for early demands. Indeed, to fulfill a demand $D(t,r)$ with $r \in [0, L_{ord}[$, the containers must be sent at period $t-1$ and can be re-ordered at period $t+1$. To fulfill a demand $D(t,r)$ with $r \in]R - L_{del}, R[$, the containers must be ordered at period t but will arrive to the manufacturer after the ordering time of period $t+1$, so they can only be re-ordered at period $t+2$. In the special case $L_{ord} = 0$, there is no preliminary time step, the early time steps are $r \in [0,\ R - L_{ord} - L_{del}]$, and the late time steps are $r \in]R - L_{ord} - L_{del},\ R[$.

The presence of non-early demands creates a challenge in deciding how many containers should be ordered at each period, because a part of the containers ordered at period t cannot be sent at period $t+1$. Even for stationary costs, constant fleet size and deterministic demand, an optimal ordering policy

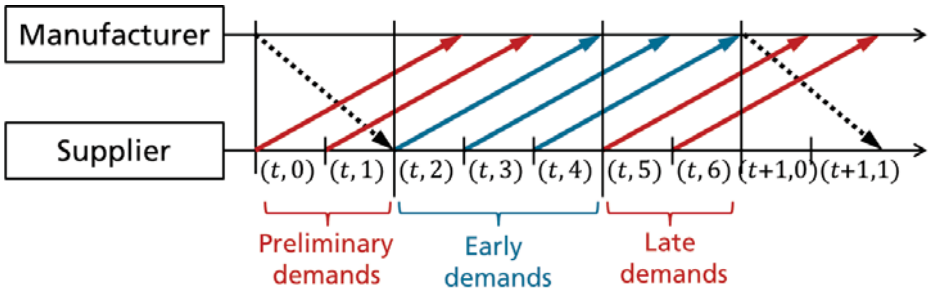


Figure 2.3: Partitioning of the demand in a period into preliminary, early and late demands, when $L_{ord} = 2$ and $L_{del} = 3$. The time is represented on the horizontal axis. The solid arrows represent the flow of full packages from the supplier to the manufacturer, whereas the dotted arrows represent the flow of empty containers.

may use a container for a late demand with low profitability at period t and a disposable for a more profitable late demand at period $t+1$, or reciprocally.

2.3.2 Lost-Sales Models

When the supplier does not have enough containers to entirely fulfill a demand, single-use disposables are bought to get enough packages. This way of dealing with the lack of packages is called a *lost-sales model*. Two alternative models are the *non-shortage model*, where the supplier must fulfill every demand using exclusively containers, and the *backlogging* model, where the unfulfilled demand is repeatedly postponed to the next period until we have enough containers. It is also possible to make combination of these strategies.
In the case of backlogging, the supplier has to pay a shortage cost at every period where containers are missing, so we will pay a shortage cost several times for the same item until there is a container to fulfill it. In the stochastic inventory control literature (see Chapter 7), it is well known that the models with backlogging are much simpler to solve than the models with lost-sales. Nevertheless, when items are returnable containers, backlogging does not seem to make the problem simpler since we also need to track the arrival of full containers at the manufacturer location. Furthermore, in Chapter 3 we formulate the deterministic version of the container purchasing problem as a network flow, whereas there is no simple way to model it under backlogging.

2.3.3 Consequences of the Setup Cost

In this thesis, we suppose that a setup cost has to be paid whenever new containers are purchased. This assumption fundamentally changes the cost structure of the process because other costs are linearly dependent on the

number of corresponding packages. A linear cost structure usually provides interesting properties helping to solve the problem. In the deterministic part, we will see that the problem without setup cost can be solved very efficiently with a minimum linear-cost flow algorithm, whereas including a setup cost makes it very complex. In the stochastic part, we will see that the linear problem has convexity properties whereas the case with setup cost does not.

2.4 The Wagner-Within Algorithm

The W-W model is a pure lot-sizing problem with no direct connection to container management. The W-W model is referred to as P_{ww}. We muss satisfy demands $D_{ww}(t)$ over a time horizon of T periods $t \in [0, T[$. However, they must be entirely fulfilled, without any backlogging or lost-sales.
There is a purchasing setup cost when purchasing items and a holding cost at the end of each period. The objective is to find a cost minimizing purchasing policy. We call *placement* a period where at least one container is purchased. We denote by $x(t)$ the manufacturer stock at the beginning of period t, before purchasing, and by $\alpha(t)$ the purchase quantity before fulfilling demand $D_{ww}(t)$. A fundamental property of the W-W model is the so called *zero inventory ordering property*:

Proposition 2.4.1 ***[Zero Inventory Ordering (ZIO) Property [128]]:*** *There is an optimal solution to P_{ww} so that the manufacturer stock is empty at the end of each period preceding a placement:*

$$\forall t : \ x(t) \cdot \alpha(t) = 0 \tag{2.11}$$

This property greatly simplifies the resolution of the problem. If we decide to purchase containers at period t, then we can divide the problem P_{ww} into two smaller problems:

1. A first sub-problem fulfilling demands up to period $t-1$.
2. A second sub-problem fulfilling demands starting from period t.

We now introduce notations for the W-W algorithm:

- $f(k)$: a policy fulfilling the demands $D(t)$ for $t \in [0, \ k[$.
- $F(k)$: cost of policy $f(k)$.
- $f(k_1, k_2)$: a policy fulfilling every demand $D(t)$, for $t \in [0, \ k_2[$ under the assumption that the last placement is at period k_1.
- $F(k_1, k_2)$: cost of policy $f(k_1, k_2)$.

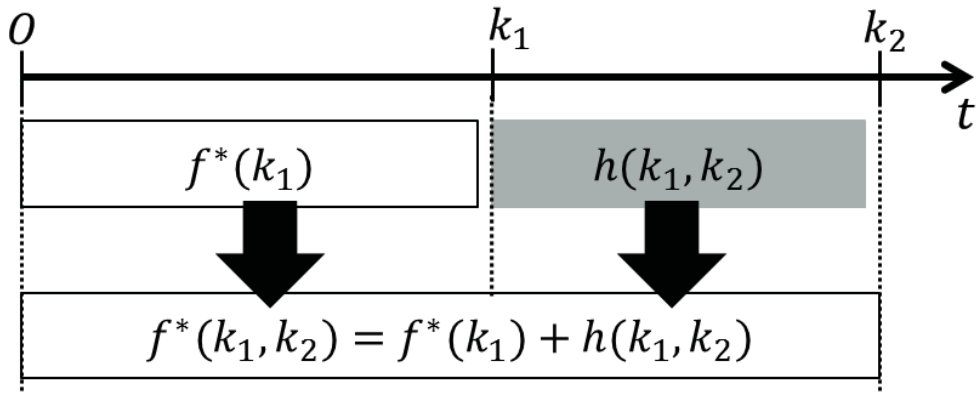


Figure 2.4: Construction of an optimal solution due to the ZIO property.

- $h(k_1, k_2)$: the unique policy fulfilling every demand $D(t)$, $t \in [k_1,\ k_2[$, under the assumption that the only placement is at period k_1.

- $H(k_1, k_2)$: cost of policy $h(k_1, k_2)$.

We say that a policy is *locally optimal* for a sub-problem if it solves it at minimum cost. For example, we say that $f(k)$ is locally optimal if it only fulfills demands from period 0 to period $k-1$ and does it at minimum cost. Clearly, if $f(T)$ is locally optimal, then it is an optimal solution to problem P_{ww}. We use the notations f^* and F^* for locally optimal solutions. We have by definition that:

$$\forall k_2 :\ F^*(k_2) = \min\{F^*(k_1, k_2);\ k_1 \in [0, k_2[\} \tag{2.12}$$

The ZIO property implies that a locally optimal solution $f^*(k_1, k_2)$ can be generated by 'adding' the purchasing policies from any locally optimal solution $f^*(k_1)$ and $h(k_1, k_2)$, as illustrated in Figure 2.4. Alternatively, if there is an optimal solution to P_{ww} respecting the ZIO property so that k is the last placement, then we can generate $f^*(T)$ by computing two locally optimal solutions $f^*(k)$ and $h^*(k, T)$, where $h^*(k, T)$ contains a single placement k and $f^*(k)$ is defined on a smaller time horizon.

The W-W algorithm generates a locally optimal solution $f^*(N)$ by computing every possible value $f^*(k_1, k_2)$. This algorithm is a dynamic program that we will use in most of our algorithms. The term *dynamic programming* refers to a framework consisting in computing a set of interdependent values in the right sequence to avoid computing the same value several times. Dynamic programming was first introduced by Bellman [9] to efficiently solve multi-stage mathematical problems.

Algorithm 1: W-W algorithm

$F(0) := 0$;
for *period* k_2 *from* 1 *to* T **do**
  $k_1 := \arg\min \left\{ F^*(k, k_2) := F^*(k) + H(k, k_2);\ k \in [0, k_2[\ \right\}$;
  $f^*(k_2) := f^*(k_1) + h(k_1, k_2)$;
return $f^*(T)$;

The complexity of this algorithm is $O(T^2)$. Nonetheless, Federgruen and Tzur [27], Wagelmans et al. [127] as well as Aggarwal and Park [1] found different approaches to solve this problem in $O(T)$ time. We will not explain these algorithms here as they are quite complex and we could not extend them to our container purchasing problem.

2.5 Notations for the Simulations

In our simulations, we randomly generate our demand and cost data. We use the binomial and uniform distributions: We denote by $\mathcal{U}(a, b)$ the uniform distribution taking any integral value between a and b included. We denote by $\mathcal{B}(n, p)$ the binomial distribution summing n Bernoulli random values, so the values go from 0 to n.
We write $X \to \mathcal{D}$ to denote that variable X follows distribution $\mathcal{D}$. In particular, in the deterministic setting our demand and cost may follow random distributions; every random value is generated before running the algorithm. This approach is used to quickly generate different scenarios.
Finally, we define the sum $\mathcal{D}_1 + \mathcal{D}_2$ of two distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ so that the random variable is the sum of two random variables following the corresponding distributions. For instance, a random variable following $\mathcal{U}(1, 3) + \mathcal{B}(1, 5)$ is generated by summing the realizations of a random variable following $\mathcal{U}(1, 3)$ and a second random variable following $\mathcal{B}(1, 5)$.

Nomenclature

General Notations

(t, r) point in time, corresponding to time step r of period t.

$:=$ Symbol of equality as a definition or affectation.

α_t Number of containers purchased at period t (decision).

$\mathbb{N}$ Set of natural numbers.

$\mathbb{R}$ Set of real numbers.

$\mathbb{Z}$ Set of integral numbers.

β_t Number of empty containers ordered at period t (decision).

$C_{cont}(t)$ Unit purchasing cost of a container at period t.

$C_{dis}(t, r)$ Cost of buying a disposable at time (t, r).

$C_{idle}(t)$ Total manufacturer holding cost during period t.

$C_{man}(t, r)$ Manufacturer holding cost at time (t, r).

$C_{setup}(t)$ Setup cost of purchasing containers at period t.

$C_{sup}(t, r)$ Supplier holding cost at time (t, r).

u_t Container fleet size at period t before purchasing.

u_t^{α} Container fleet size at period t after purchasing.

L_{del} Delivery delay of full containers from the supplier to the manufacturer.

L_{ord} Ordering delay of empty containers from the manufacturer to the supplier.

U_M Upper bound of the container fleet size.

k index used exclusively for purchasing times.

R Number of time steps per period (ex. working days per week).

r index used exclusively for time steps.

T Number of periods (ex. weeks).

t index used exclusively for periods.

CPP Container Purchasing Problem.

Notations for the Deterministic Study

a_{dis} Arc representing the use of disposables.

a_{end} Arc representing the leftover containers at the end of the time horizon.

a_{end} Arc representing leftover containers at the end of the time horizon in the networks for Algorithms $Flow$.4.1 and $Flow$.4.2.

$a_{end}^{5.1}$ Arc representing leftover containers at the end of the time horizon in the networks for Algorithm $Flow$.5.1.

$a_{end}^{5.2}$ Arc representing leftover containers at the end of the time horizon in the networks for Algorithm $Flow$.5.2.

$Flow$.4.1 First deterministic algorithm developed for increasing demand.

$Flow$.4.1 Second deterministic algorithm developed for increasing demand and allowing close placements.

$Flow$.5.1 First deterministic algorithm developed for general demand patterns.

$Flow$.5.2 Second deterministic algorithm developed for general demand patterns and allowing close placements.

a_{man} Arc representing the container holding for the manufacturer.

a_{pur} Arc representing the container purchasing.

a_{sup} Arc representing the container holding for the supplier.

$\mathcal{CA}(f)$ Container assignment on flow f.

$\mathcal{F}$ Specific subnetwork of $\mathcal{G}$.

$\mathcal{G}$ Network modeling the deterministic problem.

$\mathcal{H}$ Specific subnetwork of $\mathcal{G}$.

$\mathcal{M}cf$ Cost of an arbitratry minimum linear-cost flow algorithm.

$\mathcal{SP}$ Cost of an arbitratry shortest path algorithm.

$\mathbb{G}(t,r)$ Profitability of demand $D(t,r)$.

v_{man} Node representing the manufacturer stock at time (t,r).

v_{pur} Node representing the third party provider who produces new containers.

$v_{sup}(t,r)$ Node representing the supplier stock at time (t,r).

$D(t,r)$ Demand at time (t,r).

$D(t,r,i)$ Demand unit number i in $D(t,r)$.

$D\big[(t_1,r_1) \to (t_2,r_2)\big]$ Total demand in the time interval $[(t_1,r_1),\ (t_2,r_2)]$.

f Flow on the corresponding network $\mathcal{F}$.

f^* Minimum cost flow on the corresponding $\mathcal{F}$.

h Flow on the corresponding network $\mathcal{H}$.

h^* Minimum cost flow on the corresponding network $\mathcal{H}$.

DCPP Deterministic Container Purchasing Problem.

Notations for the Stochastic Study

$\mathbb{E}$ Expected value operator.

$\mathbb{F}$ Cumulative distribution function.

$\mathbb{P}$ Density function.

$\Delta_{t,r}$ Random variable for the number of containers used for demand $D_{t,r}$.

$\delta_{t,r}$ Realization of variable $\Delta(t,r)$.

$D_{\max}$ Upper bound of the demand variables $D_{t,r}$.

EoH End of the time horizon, after which the costs and the demands are null.

u_t Random variable for the container fleet size at period t before purchasing.

u_t^α Random variable for the container fleet size at period t after purchasing.

γ_t Online decision at period t.

ψ Expected cost of the current time or period.

φ Expected cost of the considered policy from the current time and state up to the end of the time horizon.

φ' Expected cost function.

φ^* Expected cost of an optimal policy from the current time and state up to the end of the time horizon.

$\hat{\boldsymbol{s}}_{\boldsymbol{t,r}} = [\hat{\boldsymbol{x}}_{\boldsymbol{t,r}}, \hat{\boldsymbol{y}}_{\boldsymbol{t,r}}, \hat{\boldsymbol{z}}_{\boldsymbol{t,r}}]$ Realization of the alternative state formulation.

$\boldsymbol{S_{t,r}}$ Random variable of the process state at time (t, r).

$\boldsymbol{s_{t,r}}$ Realization of variable $\boldsymbol{S_{t,r}}$.

$D_{t,r}$ Random variable for the demand at time (t, r).

$d_{t,r}$ Realization of variable $D(t, r)$.

$X_{t,r}$ Parameter of state $\boldsymbol{S_{t,r}}$.

$x_{t,r}$ Realization of variable $X_{t,r}$.

X_t Supplier stock at period t.

$Y_{t,r}$ Parameter of state $\boldsymbol{S_{t,r}}$.

$y_{t,r}$ Realization of variable $Y_{t,r}$.

Y_t Manufacturer stock at period t.

$Z_{t,r}$ Parameter of state $\boldsymbol{S_{t,r}}$.

$z_{t,r}$ Realization of variable $Z_{t,r}$.

Z_t Number of outgoing full containers at at period t.

SCPP Stochastic Container Purchasing Problem.

Part I

Deterministic Study

Chapter 3

Deterministic Model

In this chapter, we describe the deterministic container purchasing problem ($DCPP$) and model it as a minimum cost flow on a network with concave cost. Solving this flow problem is the subject of the next chapters.
Section 3.1 provides a detailed state of the art, regrouping literature on both lot-sizing and container management problems. Section 3.2 introduces the reader to network flows. Section 3.3 analyzes properties specific to the deterministic model. Finally, Section 3.4 formulates the network flow.

3.1 Literature Survey

The deterministic container purchasing problem in this thesis builds up on two main streams of research, namely lot-sizing problems and empty container repositioning problems.

3.1.1 Lot-Sizing Problems

Lot-sizing problems consist in deciding how many items we should buy at each time step to face future demands, given a manufacturing setup cost and a linear holding cost.
The research on lot-sizing is very old, and the current literature is very large. Harris [41] determines the *economic order quantity* in a logistic process with constant demand and stationary costs. This formula gives the optimal order quantity and the optimal order interval in a continuous review production system. In such a system, the time horizon is not discretized into periods and thus items can be produced at any time. In particular, this study has been extended by Donaldson [21] to a linearly increasing demand.

Dynamic Lot-Sizing Problems

A fundamental work on lot-sizing is due to Wagner and Within [128]. They introduce the *dynamic lot-sizing* problem, where the cost functions and the

demand may vary over a finite and discrete time horizon of T periods. They provide a first resolution in $O(T^2)$ time using dynamic programming (see Chapter 2). The heuristic from Silver and Meal [107] is a very popular heuristic for the dynamic lot-sizing problem to approximate at every placement the optimal purchase size and the next placement. Such an algorithm is particularly interesting when applied to a rolling horizon environment, where exact algorithms are not efficient due to the time horizon being unlimited. When the time horizon is finite, Van den Heuvel and Wagelmans [123] show however that the worst case performance guaranty of a rolling horizon heuristic is at best two, i.e. for every rolling horizon policy there exists a demand pattern and cost functions so that the policy cost is at least twice as great as an optimal policy.
Aksen et al. [3] propose an extension of the Wagner-Within (W-W) algorithm allowing lost-sales in $O(T^2)$ time. The dynamic lot-sizing problem has been widely extended to more complex scenarios, including in particular stochastic demands, product remanufacturing, production capacity constraints or perishable products. In his thesis, Van den Heuvel [122] describes and analyzes different dynamic lot-sizing variants. He proposes efficient heuristics and performance bounds for the dynamic lot-sizing problem and extensions to remanufacturing and pricing. In the following, we present extensions related to our research.

Lot-Sizing with Remanufacturing

The remanufacturing option is somehow very close to our use of containers, but differs from it by the lack of relationship between departure and arrival of the items: in lot-sizing problems with remanufacturing, it is assumed that the product returns are known in advance and do not depend on the previous demand. This means in particular that more products can return than the number that had been sent, and that backlogging and lost-sales have no impact on the returns.
Richter and Sombrutzki [94] as well as Richter and Weber [95] propose lot-sizing models with simple remanufacturing options, where we have either a purchasing option or a remanufacturing option. Golany et al. [31] and Yang et al. [134] consider a lot-sizing problem with separate decision of manufacturing and remanufacturing products and with concave costs. They formulate the problem as a network flow, prove that it is NP-hard even for stationary costs, and provide a heuristic algorithm. Teunter et al. [119] study the dynamic lot-sizing problem with linear costs and either joint or separate setup costs for manufacturing products and remanufacturing returned products. For joint costs, they prove that there the stock at the beginning of any period has at most $O(T^2)$ possible values. They deduce a $O(T^4)$ time extension of the W-W algorithm. For separate setup costs, they conjecture that the problem is NP-hard and propose an heuristic. Van den

Heuvel [121] proves that the dynamic lot-sizing model with separate setup costs is NP-hard. Helmrich et al. [46] consider and develops the previous lot-sizing models with remanufacturing. In particular, they show that the models with joint and with separate setup cost are NP-hard. Moreover, they develop and simulate efficient heuristic algorithms. Wang et al. [130] consider a lot-sizing problem with remanufacturing and outsourcing, where the outsourcing option is equivalent to lost-sales. In their setting, manufacturing and remanufacturing have separate setup costs, and they solve the problem similarly to [119]. Schulz [102] proposes an extension of the Silver-Meal heuristic to include a remanufacturing option.

Multi-Echelon Lot-Sizing

Our problem shares similarities to two-echelons lot-sizing problems. The first echelon represents the manufacturer dealing with container purchasing and packages arrival. The second echelon represents the supplier receiving empty containers and sending full packages. Zangwill [138] proposes an algorithm in $O(T^4 \cdot N)$ time to solve a lot-sizing problem on N echelons, where demands only occur in the last echelon. When $N = 2$, the complexity is improved to $O(T^3)$ by Van Hoesel et al. [124], then to $O(N^2 \cdot T^2 \cdot \log[N \cdot T])$ by Melo and Wolsey [76]. Lee et al. [67] provide an $O(T^6)$ algorithm for $N = 2$ with demand backlogging. Melo and Wolsey [77] describe heuristics for this problem. Barany et al. [7] present a polyhedral study for a multi-echelon lot-sizing problem, leading to fast algorithms which can be generalized to more complex models. The book of Pochet et al. [88] gives an review of extensions. The closest multi-echelon study to our problem is from Zhang et al. [140], who consider a lot-sizing with several echelons of demands. Indeed, for three echelons, their model consists in one echelon of purchasing and two echelons of demand. The network flow we describe in this chapter can be seen as composed of three echelons: one echelon of purchasing, one echelon of negative demand representing container arrival and one echelon of positive demand. An echelon of purchasing may be modeled as an echelon of high negative demand, where the last period of the time horizon is kept to store the not-purchased containers. If we invert the sign of the demands, our supplier echelon corresponds to the echelon of purchasing whereas our manufacturer and purchasing echelons correspond to the two echelons of demand. Zhang et al. solve a three echelons problem in $O(T^4)$ time.

3.1.2 Empty Container Repositioning

In the literature on empty container repositioning, most papers formulate either a minimum linear-cost flow or an integer program. Flow networks are typically time-expanded graphs, containing a node per location and per time period plus a few additional nodes. The arcs represent the flow of

containers between locations from one time period to a later one. One of the earliest work in container management is due to White [132]. He formulates a minimum linear-cost network flow and solved it with the out-of-kilter algorithm, which was the best algorithm at the time. This algorithm is pseudo-polynomial and not as efficient as the algorithms we use in this thesis. Crainic et al. [18] present deterministic and stochastic models for the empty container repositioning problem. They provide a general framework to solve the deterministic container management problem using network flows. The stochastic problem is modeled as a two-stage stochastic program. Gao [28] proposes an integer linear program to optimize the initial container fleet size and the container repositioning decisions in a liner shipping system. Shen and Khoong [103] describe an empty container repositioning decision system allowing the decision maker to not fulfill every demand. This lost-sales option is equivalent to our use of disposables, as in both cases the company has to pay an extra amount of money for not satisfying a part of the demand, and one less container arrives to the demand destination. Karimi et al. [61] consider a large scale container repositioning problem in the chemical industry. They assume that the container fleet size is infinite and propose an efficient linear programming formulation. Erera et al. [25] also study an intermodal container management problem for a chemical industry. The problem is formulated as minimum linear-cost flow on a time-expanded network, clustered into geographic regions. The transport between two locations in the same region is cheap and fast, whereas the transport between two different clusters is long and expensive. In another paper [24], the authors generalize their study to uncertain demands using a robust optimization approach. Jula et al. [60] study a more complex container management system where each demand must be satisfied in a time window. The decision maker owns different trucks and must give a traveling route to each truck. Olivo et al. [84] formulate a minimum linear-cost flow for empty container management. They try to fix the container imbalance in a process where each demand must be satisfied during a given time interval. To avoid infeasibility, they extend their model to add an option of leasing containers. Container leasing has an excessive cost so that it is only done when necessary. This is different from our $DCPP$ where we look for a compromise between the use of disposables and containers. Di Francesco [20] considers in his thesis several deterministic container management problems and models each of them either as a network flow or as an integer linear program. In Chandhoul et al. [12], the authors analyze a container management problem between a single manufacturer and a single supplier. They include the option to store unused containers in a depot. Contrary to the other papers, this paper assumes that the lead time between the location is immediate. The authors also assume that every demand must be fully satisfied, hence consider no lost-sales and no disposable. Imai et al. [52] compare two kinds of large scale service networks. The multi-port network

allows locations to directly exchange containers. On the other hand, the hub-and-spoke model only allows one location from each region to send and receive containers from outside of the region. Glock and Kim [30] consider a supply chain between a supplier and several clients for a constant demand. They focus on the ordering decision when the production rate is limited.

Up to this point, very few papers on empty container repositioning include a purchasing option at a setup cost, as this consideration makes the problem much more complex. The only papers we are aware of are from Chandoul et al. [13] and Moon et al. [79]. Both solve the problem using integer linear programming. In Chandhoul et al. [13], the authors consider the same model as in their previous work [12], so without lead times or disposables, and deduce some properties of the optimal solutions. Moon et al. [79] consider empty container repositioning among multiple maritime ports, with container purchasing and container leasing options. The leasing-in and leasing-off options are a simple way to manage containers when the demand is fluctuating. Containers are leased-in when the demand is too high and leased-off when the demand decreases. Containers are purchased at a high unit cost but without setup cost, and are leased at a low unit cost but with a setup cost. The objective of having two different cost structures is to make the distinction between the desired container fleet size, and the demand surges. The authors also include a setup cost for container transportation, and approximate the optimal solution with a genetic algorithm.

3.2 Introduction to Network Theory

3.2.1 Definition

Network flow problems can be described as the task of sending some commodities through a network from some locations to some others and satisfying some cost and capacity constraints. The two main flow problems are

1. the *maximum flow problem*, where we want to send as many commodities as possible from one point to another, under some restrictions on the number of items we can send through each route.

2. the *minimum cost flow problem*, where we want to send a specified number of commodities between some points at minimum cost, under some cost structure whenever we send items through a route.

In this thesis, we are exclusively interested in the minimum cost flow problem. A *flow network* is a directed graph $\mathcal{G} = (V_\mathcal{G}, A_\mathcal{G}, C_\mathcal{G}, E_\mathcal{G}, U_\mathcal{G}, L_\mathcal{G})$, where:

- $V_\mathcal{G}$ is a set of *nodes* representing some locations at a specified point in time, and

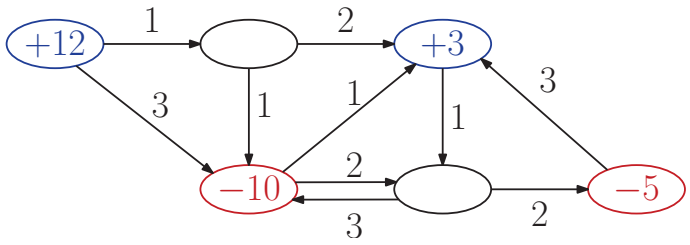


Figure 3.1: Example of a flow network. Each node is annotated with its excess value if not zero, and each arc with its linear cost. Sources are in blue while sinks are in red.

- $A_{\mathcal{G}}$ is a set of directed arcs between two nodes of $V_{\mathcal{G}}$, representing the shipping routes.

An example of a flow network is given in Figure 3.1. Each node $v \in V_{\mathcal{G}}$ is associated to an *excess* $E_{\mathcal{G}}(v)$ representing the number of commodities provided or needed. A node v with $E_{\mathcal{G}}(v) > 0$ has an excess of commodities to get rid off and is called a *source*, whereas a node with negative excess has a need of commodities and is called a *sink*. Each arc $a \in A_{\mathcal{G}}$ possesses a cost function $C_{\mathcal{G}}(a)$ so that sending x commodities over the arc incurs a cost $C_{\mathcal{G}}(a)(x)$. In addition, there may be a minimum $L_{\mathcal{G}}(a) \geq 0$ and a maximum $U_{\mathcal{G}}(a) \geq 0$ number of commodities that can be sent over any arc a. We define the *flow in an arc* a as the number of commodities sent through this arc. We define a *pseudo-flow* as a function f attributing a flow in each arc. Given an arc $a = (v_1, v_2)$ going from node v_1 to node v_2, we say that the flow of arc a *goes out of* v_1 and *goes in* (or goes to) v_2. Given a node $v \in V_{\mathcal{G}}$, we denote by $In_{\mathcal{G}}(v)$ and $Out_{\mathcal{G}}(v)$ to set of arcs going in node v and going out of node v respectively. We then define the *imbalance* of a node as its resulting excess after sending commodities according to the current pseudo-flow. Thus, the node's imbalance equals its excess plus the total flow going to it and minus the total flow going out of it. We say that a node respects the *imbalance constraint* if its total imbalance is zero. A pseudo-flow f on network $\mathcal{G}$ is called *flow* if every node respects the imbalance constraint. Given network $\mathcal{G}$, the objective of the minimum cost flow problem is to compute a flow f^* on $\mathcal{G}$ minimizing the total costs:

$$\min_{f} \sum_{a \in A_{\mathcal{G}}} C_{\mathcal{G}}(a)\left(f(a)\right) \tag{3.1}$$

s.t.

$$L_{\mathcal{G}}(a) \leq f(a) \leq U_{\mathcal{G}}(a) \qquad \forall a \in A_{\mathcal{G}}$$

$$E_{\mathcal{G}}(v) + \sum_{a \in In_{\mathcal{G}}(v)} f(a) + \sum_{a \in Out_{\mathcal{G}}(v)} f(a) = 0 \qquad \forall v \in V_{\mathcal{G}}$$

In our networks, we will always have:

$$L_{\mathcal{G}}(a) = 0 \quad \forall a \in A_{\mathcal{G}} \tag{3.2}$$

We say that a network is *uncapacitated* if

$$U_{\mathcal{G}}(a) = +\infty \quad \forall a \in A_{\mathcal{G}} \tag{3.3}$$

Network flows received a lot of attention in the scientific community in the second half of the twentieth century, and are now frequently used to solve deterministic economical problems. The following literature review covers flow networks with linear cost functions and with concave cost functions.

3.2.2 Results on Minimum Linear-Cost Flows

The minimum linear-cost networks assume that the cost of the flow in each arc is proportional to the flow quantity. These networks are simple and have very nice properties. We recommend the book of Ahuja et al. [2] for an introduction to network flows and to algorithms on linear-cost networks. Under linear costs, there is a large number of algorithms to compute a minimum cost flow, including the well known simplex algorithm and the out-of-kilter algorithm.
We distinguish two main frameworks of algorithms to solve the minimum linear-cost flow problem. In the first strategy, we start with an empty pseudo-flow and turn it into an optimal flow by iteratively adding some flow in the arcs. The second approach starts from a sub-optimal flow and improves it until it has minimum cost.
The two most interesting families of algorithms for us are the minimum cycle-canceling algorithms and the successive shortest path algorithms, as both can solve the minimum linear-cost flow problem in polynomial time [2]. In the following, we review these two approaches.

Shortest-Path-based Algorithms

Successive shortest path algorithms start with an empty pseudo-flow on the network and iteratively compute a minimum cost path from an arbitrary source to an arbitrary sink and add it to the current pseudo-flow solution. Once the imbalance property is respected for every node, the solution is a minimum linear-cost flow.
In each iteration, the path is computed using a shortest path algorithm in the so-called *residual network* relative to the current pseudo-flow solution: given a pseudo-flow f on an uncapacitated network $\mathcal{G}$, the residual network $\mathcal{R}(f)$ is a capacitated network so that any pseudo-flow f' on $\mathcal{R}(f)$ induces a feasible pseudo-flow $f + f'$ on $\mathcal{G}$ so that the cost of $f + f'$ equals the cost of f plus the cost of f'. The residual network $\mathcal{R}(f)$ has different arc costs

than $\mathcal{G}$, and an additional arc $a' = (v_2, v_1)$ for each arc $a = (v_1, v_2)$ with positive flow in f. For further details, we refer to Ahuja et al. [2].
On general networks, the Dijkstra algorithm is the fastest one and computes the shortest path from one node to every other node in $O(M + N \cdot \log[N])$ time, where N denotes the number of nodes and M the number of arcs.
Currently, the most efficient algorithm, both in theory and in practice, is the *enhanced capacity scaling* (ECS) algorithm developed by Orlin [85].

Theorem 3.2.1 ***(Orlin 1993)*** *Let $\mathcal{G}$ be an uncapacitated network with linear costs, N nodes and M arcs. Let $\mathcal{SP}$ be the time complexity to compute a shortest path from one node to every other node in $\mathcal{G}$. Then the ECS algorithm computes a minimum cost flow on $\mathcal{G}$ in $O\left(N \cdot \log[N] \cdot \mathcal{SP}\right)$ time.*

Corollary 3.2.2 ***(Orlin 1993)*** *Using the Dijkstra algorithm, the ECS algorithm solves the uncapacitated minimum linear-cost flow problem in time $O\big(N \cdot \log[N] \cdot (M + N \cdot \log[N])\big)$.*

This algorithm sends big amounts of flow at once, from sources with big excess to sinks with high demand, in order to reduce the number of shortest path computations. More precisely, it sets a scaling value Δ, and sends Δ flow units from a node with excess above $\Delta \cdot (N-1)/N$ to a node with excess below $-\Delta/N$, or from a node with excess above Δ/N to a node with excess below $-\Delta \cdot (N-1)/N$. If there are no such nodes, the scaling value is halved: $\Delta := \Delta/2$. The particularity of this algorithm compared to others is that adding a shortest path may transform a source into a sink or a sink into a source, for the sake of reducing the running time.

Cycle-Canceling-based Algorithms

An alternative way to compute minimum linear-cost flows is by using a cycle-canceling algorithm. In this framework, we define a *negative cycle* of a flow f as cycle and a direction so that:

- At least one flow unit can be added in the cycle. An arc a in the forward direction in the cycle can be incremented by up to $U_{\mathcal{G}}(a) - f(a)$ units, whereas an arc a in the backward direction in the cycle can be incremented by up to $f(a) - L_{\mathcal{G}}(a)$.
- The total cost of the directed cycle is negative, where a forward-directed arc a is affected to the unit cost $C_{\mathcal{G}}(a)$ and a backward-directed arc a is affected to the unit cost $-C_{\mathcal{G}}(a)$.

Ahuja et al.[2] show that a flow in a linear-cost network is optimal if it does not contain any negative cycle:

Theorem 3.2.3 ***(Ahuja et al. 1993):*** *In a linear-cost network, a flow is optimal if and only if it contains no negative cycle.*

Sending flow forward on a negative cycle decreases the cost of the solution. This operation is called *removing a negative cycle*. Therefore, by iteratively finding and removing negative cycles, we gradually improve the solution and end up with an optimal flow after a finite number of iterations[1].
A strongly polynomial variant of this algorithm removes cycles with minimum mean cost, i.e. so that each removed cycle minimizes the average cost per arc. In Chapter 5, we use Theorem 3.2.3 to prove that one of our operations preserves flow optimality. In Chapter 6, we develop our own cycle-canceling algorithm.

3.2.3 Literature on the Minimum Concave-Cost Flow

The minimum cost flow problem is much more complex to solve when the costs are not linear. In particular, Garey and Johnson [29] prove that finding a minimum cost flow is NP-hard when the costs have a concave structure. The result can also be found in Johnson [59].

Theorem 3.2.4 ***(Garey and Johnson 1979):*** *The minimum concave-cost flow problem is NP-hard.*

In this chapter, we formulate the deterministic container purchasing problem as a fixed-plus-linear-cost flow problem, where each arc has a cost proportional to the amount of flow in it plus a fixed value if the flow is positive. This is a special case of concave-cost networks.
One of the most important results on minimum concave-cost flows is by Zangwill [137], who provides a characterization of an optimal solution:

Theorem 3.2.5 ***(Zangwill 1968):*** *In any uncapacitated network with concave costs, there exists at least one acyclic minimum cost flow.*

Zangwill [138] studies the minimum concave cost network with a single source, and an application to lot-sizing problems. Veinott [126] analyzes a class of minimum concave-cost networks and some applications in logistics. Graves and Orlin [33] study some instances of concave-cost networks and apply their algorithm for the lot-sizing problem with fixed-plus-linear costs. Erickson et al. [26] propose a minimum concave-cost flow algorithm that is polynomial in the number of nodes and arcs, but exponential in the number of non-zero excess nodes. In the coming chapter, our formulation has a linear number of non-zero excess nodes. Therefore this algorithm is not suited for our network. Guisewite and Pardalos [37] overview minimum concave-cost networks and show the NP-hardness of some special cases. Guisewite and Pardalos [38] provide a strongly polynomial algorithm for the single-source uncapacitated minimum concave-cost flow in a network with a single

[1] This is because a negative cycle removes at least one cost unit and any flow has a non-negative cost.

non-linear-cost arc. Aggarwal and Park [1] use the data structure of Monge arrays to improve the complexity bound of some lot-sizing problems. In particular, they improve the algorithms of [33] and [26]. Recently, He et al. [44] have proven that the minimum concave-cost flow problem is polynomial if the network has a grid structure. In our case, the flow network would have a grid structure if there was no lead time or no disposables. This result is mostly of theoretical importance, due to the excessively large time complexity of the algorithm. A more complete study has then been proposed in the thesis of He [43].

3.3 Problem Analysis

3.3.1 Use of Disposables

We stated in Chapter 2 that the supplier buys disposables when he does not have enough containers for the demand. Under deterministic demand, it is debatable whether the use of disposables is meaningful or not. Indeed, disposables are usually considered to face a random demand with a high variance so that we can order a reasonable number of containers while being able to satisfy an unexpected demand surge. Shen and Khoong [103] firstly consider the problem without lost-sales, and then include a lost-sales option. Meanwhile, Di Francesco [20] as well as Chandhoul et al. [12, 13] assume that the demand cannot be lost or backlogged.
In this thesis, our main objective is to compute a solution allowing disposables. Nevertheless, the setting without disposables is much simpler and is solved in Chapters 4 and 5 respectively for an increasing demand and under a general demand pattern. Afterward, we compare the performance of the two alternatives in Chapter 6.
We expect an optimal policy to buy some disposables during the few last periods before a placement, since it is intuitively the point where we may be lacking containers in the system.

3.3.2 Ordering Delay

We simplify $DCPP$ using the following lemma:

Lemma 3.3.1 *Under deterministic demand, every instance of $DCPP$ is equi-valent to an instance with zero ordering delay.*

Proof:

Consider a problem with $L_{ord} > 0$ and $L_{del} \leq R - L_{ord}$. We can re-index the time for the supplier with $(t, r)' := (t, r + L_{ord})$. The empty containers ordered from the manufacturer at time $(t, 0)$ arrive to the supplier at time $(t, 0)'$, and the loaded parts departing from the supplier at time $(t, r)'$ arrive

to the manufacturer at time $(t, r + L_{ord} + L_{del})$. Consequently, the system with ordering time L_{ord} and delivery time L_{del} is equivalent to a system with ordering time 0 and delivery time $L_{ord}+L_{del}$, where the demands arrive L_{ord} time steps earlier.

□

Starting from now and until the end of the deterministic study, we assume that $L_{ord} = 0$, i.e. that the ordered empty containers instantaneously arrive to the supplier. This result does not hold for stochastic demands because, when we must take the ordering decision, we do not know the exact leftover supplier stock at order arrival.
Assuming $L_{ord} = 0$ simplifies our notations as we now only have early steps $r \in [0, R - L_{del}]$ where full containers arrive during the same week, and late steps $r \in [R - L_{del} + 1, R - 1]$ where full containers only arrive after the next ordering time. We have no preliminary time step anymore.

3.3.3 Supplier Stock Holding

The following lemma is a direct consequence of Lemma 2.2.1 in case of a deterministic demand.

Lemma 3.3.2 *Suppose that Hypothesis 2.2 [Cost] holds. In every optimal deterministic policy and for each period t, the supplier stock is empty after satisfying demand $D(t, R - 1)$.*

Proof:

Suppose that the supplier still has a container after satisfying demand $D(t, R - 1)$ at some period t. This container has been ordered at period t or before. By Lemma 2.2.1, ordering it at period $t + 1$ instead decreases the solution cost.

□

Corollary 3.3.3 *Under Hypothesis 2.2 [Cost] and in every optimal deterministic policy, any ordered container returns to the manufacturer within two periods of time.*

3.3.4 Maximum Fleet Size

By Corollary 3.3.3 we can bound the maximum fleet size with the total demand within a two periods interval. We denote by $D\big[(t, r) \to (t', r')\big]$ the total demand from time (t, r) up to time (t', r').

Lemma 3.3.4 *Suppose that Hypothesis 2.2 [Cost] holds. Then, in the system we will never need more than U_M containers, where:*

$$U_M := \max_{t\in[0,T[} \left\{ \sum_{r=1-L_{del}}^{R-1} D(t,r) \right\} \tag{3.4}$$

Proof:

The optimal fleet size is bounded by the maximum number of containers that we may need at any decision time $(t,0)$. During period $t \in [0, T-1]$, the number of outgoing containers is bounded by the total late demand of the previous period, from time $(t-1, R-L_{del}+1)$ to $(t-1, R-1)$. In addition, by Lemma 3.3.2, an optimal policy never orders more than $\sum_{r=0}^{R-1} D(t,r)$ containers during period t. Therefore, at the beginning of any period t we need at most than U_M containers in the system.

□

3.3.5 Integer Linear Program

The deterministic problem can be formulated as an Integer Linear Program (ILP). We denote by $\delta(t,r)$ the number of containers used to fulfill demand $D(t,r)$. We recall that by Hypothesis 2.2 the disposables must be directly used and cannot be held in stock. We define $\gamma(t) := \mathbb{1}\big[\alpha(t) > 0\big]$, and denote by $X_{end}(t,r)$ and $Y_{end}(t,r)$ the supplier and the manufacturer stock at the end of time (t,r). In the integer linear program, the variables are one or two dimensional vectors indexed with characters '[ ]' instead of '( )':

$$\begin{aligned}
\min_{\alpha,\beta,\delta,\gamma} \sum_{t=0}^{T} \Big[& \gamma[t] \cdot C_{setup}(t) + \alpha[t] \cdot C_{cont}(t) \\
& + \sum_{r=0}^{R} \Big(X_{end}[t,r] \cdot C_{sup}(t,r) + Y_{end}[t,r] \cdot C_{man}(t,r) \\
& - \delta[t,r] \cdot C_{dis}(t,r) \Big) \Big]
\end{aligned} \tag{3.5}$$

s.t.

$$\begin{aligned}
& Y_{end}[0,-1] = 0 \\
& Y_{end}[t,0] = Y_{end}[t,-1] + \alpha[t] - \beta[t] + \delta[t,-L_{del}] \\
& Y_{end}[t,r] = Y_{end}[t,r-1] + \delta[t,r-L_{del}], && \forall r \in]0,R[\\
& X_{end}[t,0] = \beta[t] - \delta[t,0] \\
& X_{end}[t,r] = X_{end}[t,r-1] - \delta[t,r], && \forall r \in]0,R[\\
& \delta[0,-L_{del}] = 0 \\
& \delta[t,0] \leq \beta[t]
\end{aligned}$$

$$\delta[t,r] \leq X[t,r-1], \qquad \forall r \in]0,R[$$
$$\delta[t,r] \leq D(t,r)$$
$$\beta[t] \leq \alpha[t] + Y_{end}[t,-1] + \delta[t,-L_{del}]$$
$$\gamma[t] \leq 1$$
$$\alpha[t] \leq U_M \cdot \gamma[t]$$
$$\alpha[t], \beta[t], \delta[t,r], \gamma[t] \geq 0$$
$$\alpha[t], \beta[t], \delta[t,r], \gamma[t] \in \mathbb{N}$$

3.4 Flow Network Formulation

We formulate $DCPP$ as a minimum fixed-plus-linear-cost flow, as illustrated in Figure 3.2. The above part shows a general description of the network while the below part indicates the linear-costs and the excess values for a representative subnetwork. We later analyze arc capacities in the network.

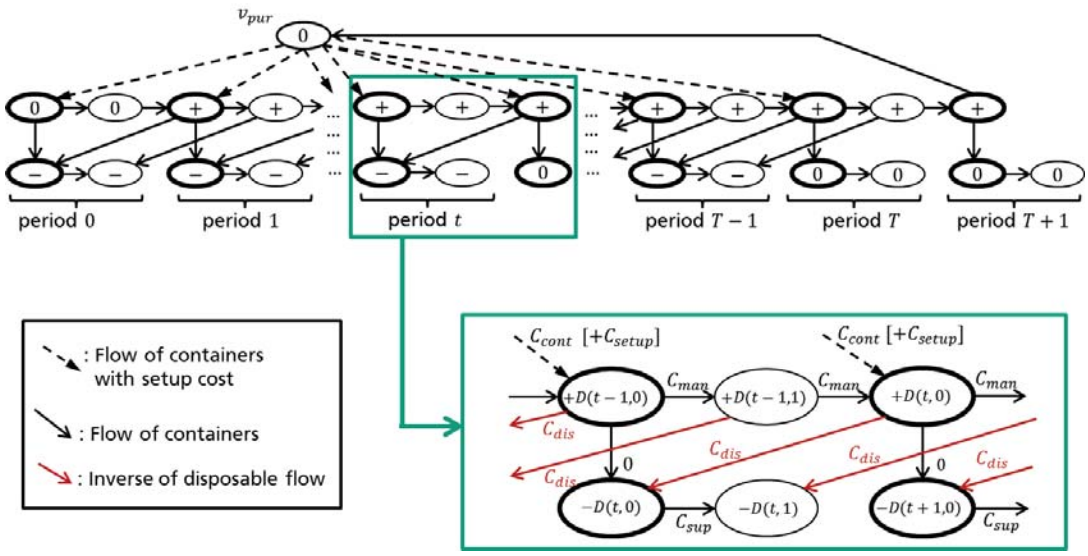


Figure 3.2: Flow network in the case $R = L_{del} = 2$ and for stationary costs. Dashed arcs have in addition a setup cost. Each node is annotated with either the excess value or its sign.

Even though demand is only between periods 0 and $T-1$, we add two dummy periods T and $T+1$ for technical reasons. There is a manufacturer node and a supplier node for each time step of each period. Bold nodes annotate the beginning of each period. Horizontal arcs represent the holding of containers. Vertical arcs describe container orderings, assumed instantaneous. Dashed arcs are relative to container purchasing, and are affected with the setup cost. Each manufacturer node has an excess equal to the number of containers departing L_{del} time steps ago, and is linked to the corresponding demand

with a backward arc. The flow in every backward arc represents the number of disposables used for the corresponding demand: for each flow unit, one less container is required for the demand and one less container arrives to the manufacturer.
Formally, we model the deterministic container purchasing problem as a network $\mathcal{G} = (V_\mathcal{G}, A_\mathcal{G}, C_\mathcal{G}, E_\mathcal{G})$, where V_G is a set of $R \cdot (2 \cdot T + 3) + 2$ nodes, $A_\mathcal{G}$ is a set of $(T + 1) \cdot (3 \cdot R + 1) + 1$ arcs, $C_\mathcal{G}$ is a cost function for the arcs and $E_\mathcal{G}$ is an excess function for the nodes. The set of nodes $V_\mathcal{G}$ contains:

1. a supplier node $v_{sup}(t, r)$ for each $t \in [0, T]$, $r \in [0, R[$, with excess $-D(t, r)$ if $t < T$ and 0 otherwise.

2. a manufacturer node $v_{man}(t, r)$ for each $t \in [0, T]$, $r \in [0, R[$ and for $(t, r) = (T + 1, 0)$, with excess:

   - 0 if $t = 0$ and $r < l'$,
   - $+D(t, r - l')$ if $t \in [0, T[$ and $r \in [l', R[$,
   - $+D(t - 1, R + r - l')$ if $t \in [0, T]$ and $r \in [0, l'[$
   - 0 if $t = T$, $r \in [l', R[$ or if $(t, r) = (T + 1, 0)$.

3. a purchasing node v_{pur} with excess value zero.

The set of arcs $A_\mathcal{G}$ contains:

1. An arc $a_{sup}(t, r)$ for each $t \in [0, T + 1]$ and each $r \in [0, R - 2]$, representing container holding at the supplier location. We recall that for each period, the stock is empty at the end of the last time step $R - 1$, so we have no arc $a_{sup}(t, R - 1)$. Arc $a_{sup}(t, r)$ has linear-cost $C_{sup}(t, r)$ and goes from $v_{sup}(t, r)$ to $v_{sup}(t, r + 1)$.

2. An arc $a_{man}(t, r)$ for each $t \in [0, T]$ and $r \in [0, R[$. Arc $a_{man}(t, r)$ models a container holding for the manufacturer and has linear-cost $C_{man}(t, r)$ if $t < T$ and no cost if $t = T$. It goes from $v_{man}(t, r)$ to:

   - $v_{man}(t, r + 1)$ if $r < R - 1$,
   - $v_{man}(t + 1, 0)$ if $r = R - 1$.

3. An arc $a_{pur}(t)$ for each $t \in [0, T]$ representing the purchase quantity at period t. Arc $a_{pur}(t)$ goes from v_{pur} to $v_{man}(t, 0)$ and has linear-cost $C_{cont}(t)$ and setup cost $C_{setup}(t)$. Arc $a_{pur}(t)$ has no cost.

4. An arc $a_{ord}(t)$ for each $t \in [0, T + 1]$ representing the order quantity at period t. Arc a_{ord} has no cost and goes from $v_{man}(t, 0)$ to $v_{sup}(t, 0)$.

5. An arc $a_{dis}(t, r)$ with linear cost $C_{dis}(t, r)$ for each $t \in [0, T[$ and each $r \in [0, R[$, representing the number of disposables used for $D(t, r)$, thus the capacity of arc $a_{dis}(t, r)$ is $D(t, r)$.

- If $r < R - l'$, then it goes from $v_{man}(t, r + l')$ to $v_{sup}(t, r)$,
- otherwise, it goes from $v_{man}(t + 1, r + l' - R)$ to $v_{sup}(t, r)$.

6. An arc $a_{end}(T + 1)$ from $v_{man}(T + 1, 0)$ to v_{pur}, with no cost.

The dummy nodes and arcs in $\mathcal{G}$ for period $t \geq T$ have no influence on the solution. Actually, the manufacturer dummy nodes are needed for disposable arcs relative to late demands of period $t-1$. It is easy to see that any solution can be written as a flow on $\mathcal{G}$. However, not every flow corresponds to a policy, since according to our network it is possible to send more than $D(t, r)$ flow units over arc $a_{dis}(t, r)$.

Lemma 3.4.1 *With capacity constraint of $D(t, r)$ to arc $a_{dis}(t, r)$, the network formulates DCPP.*

Proof:

With the capacity constraint, the network is readily equivalent to the integer linear program described in the previous section.

□

By Lemma 3.4.1, in order to prove the correctness of our network formulation, we only have to prove that in any minimum cost flow in $\mathcal{G}$, the flow in arc $a_{dis}(t, r)$ is at most $D(t, r)$. This is easily proven for stationary costs:

Proposition 3.4.2 *Suppose that Hypothesis 2.2 [Cost] holds. Then, for any minimum cost flow and any time (t, r), the flow in arc $a_{dis}(t, r)$ is at most $D(t, r)$.*

Proof:

Suppose that the flow in arc $a_{dis}(t, r)$ is greater than $D(t, r)$, and let $v_{man}(t', r')$ be the node incident to $a_{dis}(t, r)$ such that (see Figure 3.3):

$$(t', r') = (t, r) + L_{del}$$

The excess of $v_{sup}(t, r)$ is $-D(t, r)$ and the ingoing flow is at least $D(t, r)+1$. This is impossible for $r = R - 1$ because $v_{sup}(t, R - 1)$ has no outgoing arc. For $r < R - 1$, it follows that the outgoing flow in $a_{sup}(t, r)$ is positive. By Hypothesis 2.2 [Cost], sending a flow unit from $v_{man}(t', r')$ to $v_{sup}(t, r + 1)$ is cheaper via node $v_{man}(t', r' + 1)$ than $v_{sup}(t, r)$. Thus the flow is not optimal.

□

Finally, we generalize our network to hold under a weaker assumption than Hypothesis 2.2 [Cost]. Firstly, if the result from Lemma 3.3.2 does not hold, we only need a new arc $a_{sup}(t, R - 1)$ from $v_{sup}(t, R - 1)$ to $v_{sup}(t + 1, 0)$. Secondly, it may be profitable to buy a disposable while holding a container. Our network can still be generalized under the assumption that at every time

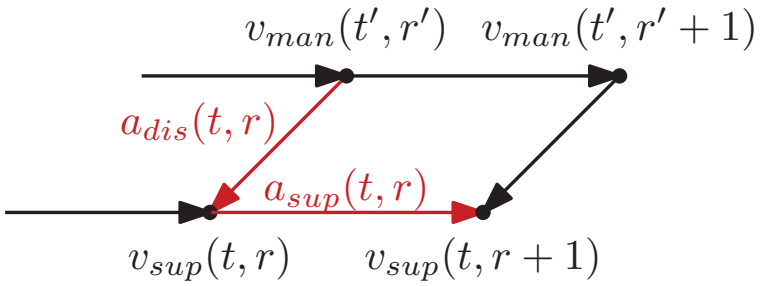
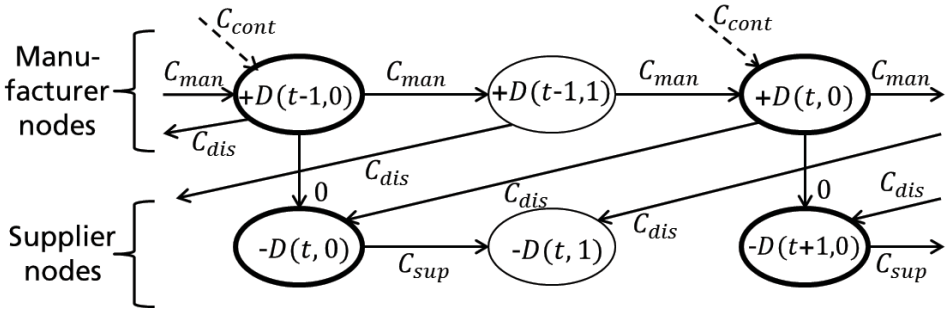
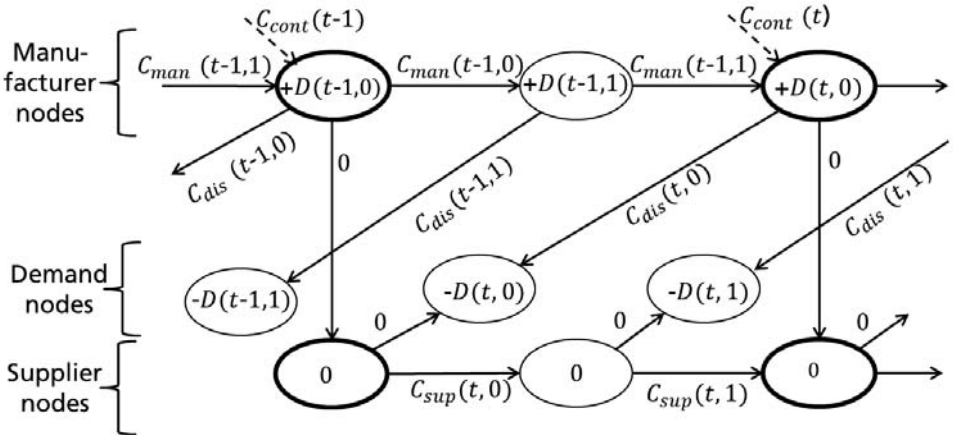


Figure 3.3: Network in the neighborhood of $a_{dis}(t, r)$. The red arcs have a positive flow.

(t, r) we are not allowed to buy more than $D(t, r)$ disposables. To avoid having infeasible flows, we need a capacity of $D(t, r)$ to arc $a_{dis}(t, r)$. Using the operation described in Ahuja et al. [2], we make the network uncapacitated again by adding $O(T \cdot R)$ nodes and arcs. Figure 3.4 illustrates the periodic pattern in the initial network $\mathcal{G}$ and in the adapted network. To generate the uncapacitated network, we separate every node $v_{sup}(t, r)$ into two nodes $v_{sup}(t, r)$ with zero excess and $v_{dem}(t, r)$ with excess $-D(t, r)$. The nodes



(a) Network pattern under Hypothesis 2.2 [Cost].



(b) Updated pattern.

Figure 3.4: Patterns of the uncapacitated flow networks, depending on whether Hypothesis 2.2 [Cost] holds or not.

$v_{sup}(t,r)$ still represent the stock of the supplier while the nodes $v_{dem}(t,r)$ correspond to the transportation of material parts from the supplier to the manufacturer. The arcs $a_{sup}(t,r-1)$ and $a_{sup}(t,r)$ are incident to node $v_{sup}(t,r)$, as well as arc a_{ord} when $r=0$. However, arc $a_{dis}(t,r)$ is now incident to $v_{dem}(t,r)$. In addition, we create new arcs $a_{cont}(t,r)$ from $v_{sup}(t,r)$ to $v_{dem}(t,r)$ with zero cost. This network update increases the number of nodes and the number of arcs by $R\cdot(T+2)$. Therefore, we still have $O(R\cdot T)$ nodes and arcs, even though the number of nodes has increased by roughly 33%.

3.5 Outlook

This chapter describes the deterministic version of the container purchasing problem in a closed-loop supply chain between a supplier and a manufacturer. Despite the literature on deterministic container repositioning and on dynamic lot-sizing being very abundant, there are no models considering the management of containers in a ramp-up environment with increasing demand. Container repositioning problems usually assume either a stable demand, or a possibility to lease-on and -off containers very easily from a third party privider and without setup cost. Such a model is very fitting for container management problems in maritime ports, where the total demand is huge and supply companies store large amounts of these *Twenty-foot Equivalent Units* containers (TEU) used between ports. However, In local closed-loop supply chains, items may require a special care hence the use of special containers. It becomes more difficult to get new containers and a more careful study is required.
Moreover, we have proven that we can assume without loss of generality in a deterministic environment that the ordering delay is zero. We have shown that in most scenarios we can assume without loss of generality that every demand is profitable. Afterward, we have formulated the deterministic container purchasing problem as a minimum cost flow on an uncapacitated network with fixed-plus-linear cost. This problem is known to be NP-hard in general, and the next two chapters are dedicated to computing a solution. Finally, we generalized the network formulation to hold for a weaker hypothesis on the cost structure.

Chapter 4

Algorithms for Increasing Demand

In a previous publication [54], we solved the deterministic problem under several restrictive assumptions. The contribution of this paper to the literature is a first polynomial time algorithm on $DCPP$, as other authors (Chandoul et al. [13], Moon et al. [79]) only write an integer linear program which they then solve using a meta-heuristic.
We present and develop these results in this Chapter. Section 4.1 describes our hypotheses and summaries our paper. Section 4.2 solves the special case where no disposable is allowed. Since the demand follows a specific pattern, the demand cannot be assumed without loss of generality to be profitable anymore. This topic is analyzed in Section 4.3. In Section 4.4, we add another hypothesis to solve the problem optimally with a first algorithm. Section 4.5 relaxes this hypothesis using the succinct hints we gave in [54], and deduce a slightly more complex algorithm. Both algorithms run in $O(R^2 \cdot T^4 \cdot \log[R \cdot T]^2)$ time. Section 4.6 presents new work, as we extend the algorithms to more general settings, namely longer lead times, several suppliers and making our algorithm also compute a feasible solution for general demand patterns.

4.1 Introduction

In this chapter, we assume that the demand $D(t, r)$ is non-decreasing with respect to t for a fixed time step r:

Hypothesis 4.1 ***[Demand]:***

$$\forall r \in [0, R[, \forall t \in [1, T[: \;\; D(t, r) \geq D(t-1, r) \tag{4.1}$$

This is a quite restrictive assumption, since in real life scenarios the demand may punctually decreases despite a clear increasing trend. We consider gen-

eral demand patterns in Chapter 5. We recall that by assumption, Hypothesis 2.1 [Delay] and 2.2 [Cost] hold. Moreover, every time the manufacturer purchases new containers, we suppose that the resulting container fleet is big enough so that the supplier does not need to buy any disposable for the next two periods:

Hypothesis 4.2 ***[Placement]:*** *There is an optimal policy so that if k is a placement, then no disposable is bought during periods k and $k+1$.*

This assumption is reasonable because when the manufacturer purchases new containers, he whould get at least enough containers for a short time horizon. Under these assumptions, we solve optimally the flow network $\mathcal{G}$ described in Chapter 3, which we recall in Figure 4.1.

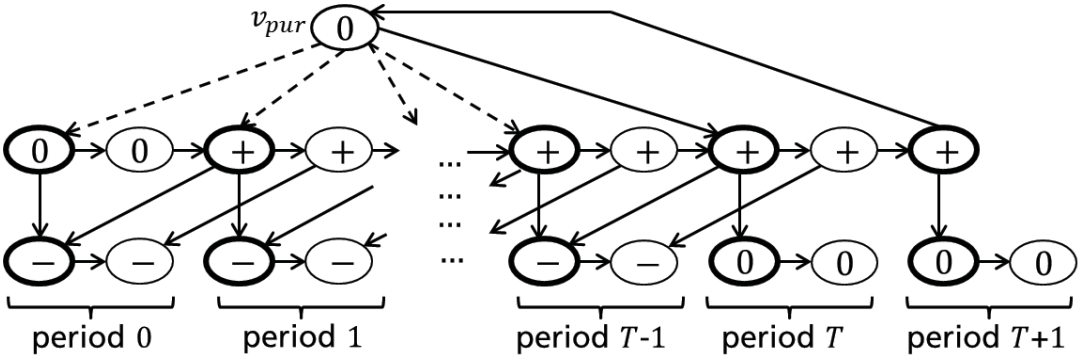


Figure 4.1: Network flow to solve in this chapter.

In Jami et al. [54], we model the problem with the same network $\mathcal{G}$. We combine the W-W algorithm [128] with a minimum linear-cost flow algorithm from the literature [85] to solve $DCPP$ optimally in $O(R^2 \cdot T^4 \cdot \log[R \cdot T]^2)$ time.

4.2 Solution without Disposables

We first consider the special case without disposables.

Hypothesis 4.3 ***[NoDisposable]:*** *Disposables are not allowed in the supply chain or are too expensive to be bought.*

In the following, we reduce the $DCPP$ to the dynamic lot-sizing model described in Chapter 2.

4.2.1 Network for the Lot-Sizing Problem

We show in Figure 4.2 a flow network formulation of the dynamic lot-sizing problem, which we denote by $\mathcal{G}_{ww}$.

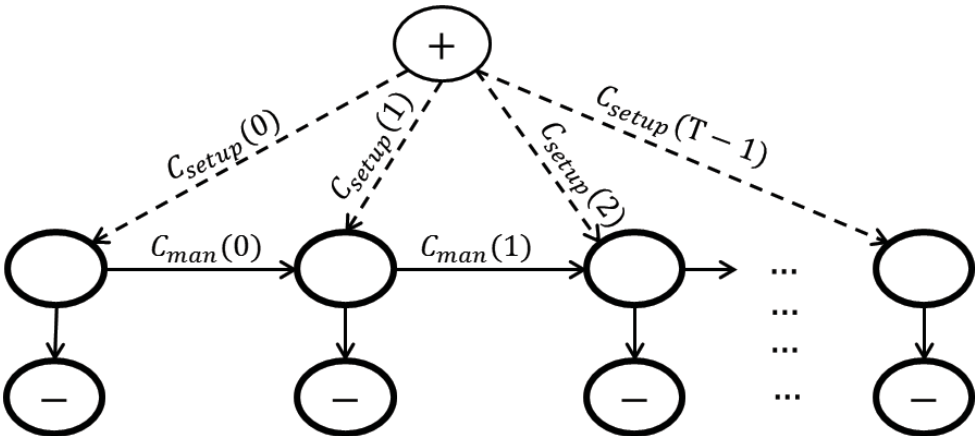


Figure 4.2: Flow network $\mathcal{G}_{ww}$ modeling the dynamic lot-sizing problem.

The bottom echelon of the network represents the supplier with a demand at each period, while the middle echelon represents the manufacturer with a holding cost and zero ordering cost. Since we are handling non-returnable items instead of containers, the manufacturer node has no excess. It is straightforward to check that the network $\mathcal{G}_{ww}$ describes the dynamic lot-sizing problem. The networks $\mathcal{G}_{ww}$ and $\mathcal{G}$ only differs on four points:

1. In $\mathcal{G}_{ww}$, there is no disposable arc.
2. In $\mathcal{G}_{ww}$, the manufacturer nodes have no excess.
3. In $\mathcal{G}_{ww}$, they use $R = 1$.
4. In $\mathcal{G}$, we require the demand to be non-decreasing.

4.2.2 Problem Reduction

Lemma 4.2.1 *Under Hypotheses 2.1 [Delay], 2.2 [Cost], 4.1 [Demand] and 4.3 [NoDisposable], DCPP and P_{ww} are equivalent.*

Proof:

We prove that we can transform $\mathcal{G}$ into a network with the same form as $\mathcal{G}_{ww}$, i.e. with the four differences we presented above.

1. *Disposable arcs:*

   Under Hypothesis 4.3 [NoDisposable], the disposable arcs in $\mathcal{G}$ are always empty, so they can be removed.

2. *Manufacturer excess:*

   A container used for an early demand at period t can always be ordered at period $t+1$. Since demand is non-decreasing and we can arbitrarily decide which container we order. We force the containers used for

an early demand $D(t,r)$ to be used for the early demand $D(t+1,r)$, and force the containers used for a late demand $D(t,r)$ to be used for the late demand $D(t+2,r)$. Then, the movements of a container are characterized by its purchasing time and the first demand it fulfills. Consequently, everything occurs as if container were not returning and we fulfilled the demands $D_{ww}(t,r)$ so that:

$$\forall t, \forall r \in [0, R - L_{del}] : D_{ww}(t,r) := D(t,r) - D(t-1,r)$$
$$\forall t, \forall r \in]R - L_{del}, R[: D_{ww}(t,r) := D(t,r) - D(t-2,r)$$

3. *Single Time Step*:

   Since the demands $D_{ww}(t,r)$ are non-returnable items and every demand must be entirely fulfilled, we can group them into a single demand $D_{ww}(t)$ per period:

$$\forall t : \ D_{ww}(t) := \sum_{r=0}^{R-1} D_{ww}(t,r)$$

4. *Dynamic Demand*:

   Despite the demands $D(t,r)$ being non-decreasing with respect to r, the equivalent demands $D_{ww}(t)$ can already take any non-negative value.

Finally, under the hypotheses of the lemma, any instance of $DCPP$ can be transformed into an instance of P_{ww}, where the demand $D_{ww}(t)$ at a period t is the sum of increase of the early demands compared to period $t-1$ and increase of the late demands compared to period $t-2$.
Meanwhile, any instance of P_{ww} can be transformed into an instance of $DCPP$ with any number R of time steps by partitioning the demands $D_{ww}(t)$ into R quantities $D_{ww}(t,r)$. The demands $D(t,r)$ for $DCPP$ are then:

$$\forall t, \ \forall r : \ D(t,r) := \sum_{x=0}^{t} D_{ww}(x,r)$$

The solution of this problem under the lemma hypotheses is also a solution to the instance of P_{ww}.

□

Proposition 4.2.2 *Under Hypotheses 2.1 [Delay], 2.2 [Cost], 4.1 [Demand] and 4.3 [NoDisposable], DCPP can be solved in $O(R \cdot T)$ time.*

Proof:

The transformation described in the proof of Lemma 4.2.1 can be done in $O(R \cdot T)$ time, and the resulting problem is solved in $O(T)$ time using for example the algorithm of Wagelmans et al. [127].

□

4.3 Demand Profitability

In Chapter 2, we noted that without loss of generality, we can assume that every demand is profitable enough so that we should fulfill it rather than keeping an empty container idle during its rotation:

$$\forall r \in [0, L_{ord}[\colon C_{dis}(t,r) + \sum_{r'=0}^{R+r+L_{del}-1} C_{man}(t-1,r') - \sum_{r'=L_{ord}}^{R+r-1} C_{sup}(t-1,r') > 0$$

$$\forall r \in [L_{ord}, R[\colon C_{dis}(t,r) + \sum_{r'=0}^{r+L_{del}-1} C_{man}(t,r') - \sum_{r'=L_{ord}}^{r-1} C_{sup}(t,r') > 0$$

Our argument is that we can first solve the simpler problem considering only profitable demand, and then satisfy unprofitable demands using disposables. However, in this chapter, the demand is assumed to be non-decreasing. Removing an unprofitable demand $D(t,r)$ from the system will violate the increasing demand assumption if a previous demand $D(t',r)$ is positive and profitable, for $t' < t$.

Lemma 4.3.1 *If the cost functions C_{man}, C_{sup} and C_{dis} are constants, then the profitability of demand $D(t,r)$ is independent from $t \in [0,T[$.*

Proof:

This follows from the definition of the profitability.

□

Corollary 4.3.2 *If the cost functions C_{man}, C_{sup} and C_{dis} are constants, then we can assume without loss of generality that every demand is profitable.*

Proof:

If a demand $D(t,r)$ is not profitable, then every demand $D(t',r)$ is not profitable, so we can remove all these demands from the process without violating Hypothesis 4.1.

□

In this chapter, we assume that every demand is profitable, even if the cost structure is not stationary.

4.4 Solution Forbidding Close Placements

We now consider a setting where disposables are allowed. To simplify the resolution, we also assume that the placements must be relatively distant from each other and that the time horizon begins with a placement and ends right before a placement. We say that two placements are *close* if they are at consecutive periods k and $k+1$. On the opposite, two placements k_1 and $k_2 > k_1$ are *consecutive* if there is no placement on interval $]k_1, k_2[$.

Hypothesis 4.4 ***[Distance]****: There is an optimal policy respecting Hypothesis 4.2 [Placement] so that:*

1. *$t = 0$ is a placement,*
2. *$t = T - 1$ is not a placement,*
3. *there are no close placements.*

We relax this hypothesis in the next section. In the following, we first generalize the notations h and f from the W-W algorithm. Then we state Proposition 4.4.1 adapting the zero inventory property, i.e. Proposition 2.4.1, to our $DCPP$. Finally, we present our algorithm.

4.4.1 Generalization of the Notations

For $k \in [0, T]$, we denote by $\mathcal{F}(k)$ the restriction of network $\mathcal{G}$ to demands of periods 0 to $k+1$, so that:

1. period k is a placement,
2. period $k+1$ is not a placement,
3. no disposable is used for periods k and $k+1$.

Network $\mathcal{F}(k)$ is generated from $\mathcal{G}$ by the following steps (see Figure 4.3).

1. Remove $v_{man}(t, r)$ for each $t > k+1$, $r \in [0, R[$ and for $t = k+1$, $r \in [1, R[$.
2. Remove $v_{sup}(t, r)$ for each $t > k+1$, $r \in [0, R[$.
3. Add a new arc $a_{end}(k+1)$ from $v_{man}(k+1, 0)$ to v_{pur} with cost 0.
4. Update the excess of v_{pur} so that the total excess in the network is 0.
5. Remove the setup cost $C_{setup}(k)$ from $a_{pur}(k)$.
6. Remove $a_{pur}(k+1)$.
7. Remove $a_{dis}(t, r)$ for $t \in [k, k+1]$, $r \in [0, R[$.

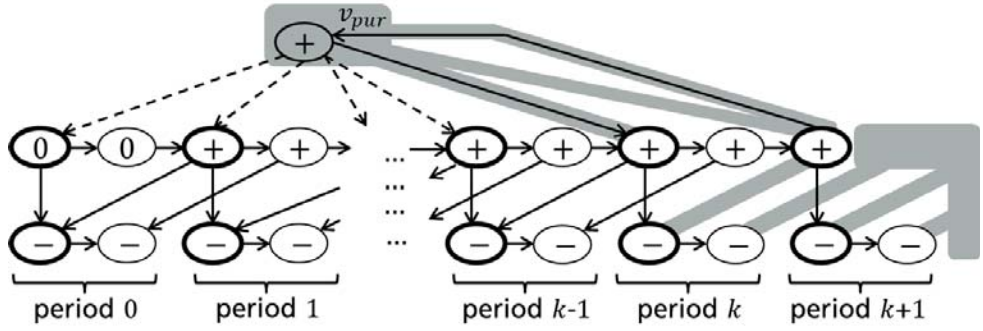


Figure 4.3: Network $\mathcal{F}(k)$. Differences from $\mathcal{G}$ are in gray.

By construction, $\mathcal{G}$ is equivalent to $\mathcal{F}(T)$ because dummy periods T and $T+1$ have been added. Let $k_1 \in [0, T-2]$ and $k_2 \in [k_1+2, T]$.
We define network $\mathcal{F}(k_1, k_2)$ from $\mathcal{F}(k_2)$ by (see Figure 4.4):

1. removing arcs $a_{pur}(t)$ for $t \in [k_1+1, k_2[$,
2. removing the setup cost on $a_{pur}(k_1)$.

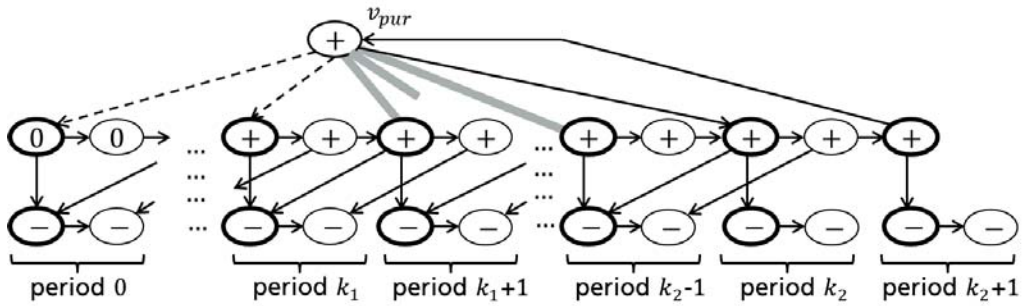


Figure 4.4: Network $\mathcal{F}(k_1, k_2)$. Differences from $\mathcal{F}(k_2)$ are in gray.

Thus, k_1 and k_2 are the last two placements in $\mathcal{F}(k_1, k_2)$. Finally, we want to define network $\mathcal{H}(k_1, k_2)$ as "$\mathcal{F}(k_1, k_2) - \mathcal{F}(k_1)$". We generate $\mathcal{H}(k_1, k_2)$ from $\mathcal{F}(k_1, k_2)$ with the following steps (see Figure 4.5):

1. Remove $v_{man}(t, r)$ for each $t \in [0, k_1[$ and $r \in [0, R[$.
2. Give zero excess to $v_{man}(t, r)$ for $t = k_1$, $r \in [0, R[$ and for $(t, r) = (k_1+1, 0)$.
3. Remove $v_{sup}(t, r)$, $a_{sup}(t, r)$, $a_{dis}(t, r)$ and $a_{ord}(t)$ for each $t \in [0, k_1+1]$ and $r \in [0, R[$.
4. Update the excess of node v_{pur} so that the total excess in the network is zero.

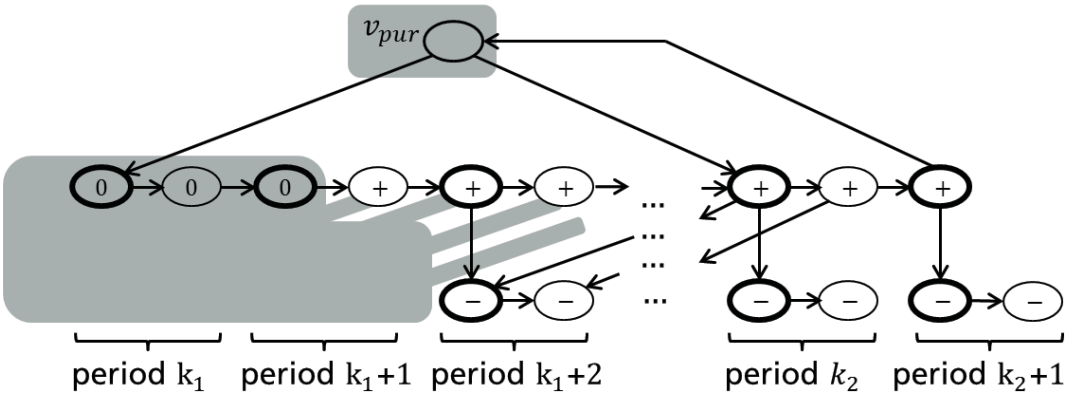


Figure 4.5: Network $\mathcal{H}(k_1, k_2)$. Differences to $\mathcal{F}(k_1, k_2)$ are in gray.

We denote by $f(k)$, $f(k_1, k_2)$, $h(k_1, k_2)$ the flows on networks $\mathcal{F}(k)$, $\mathcal{F}(k_1, k_2)$, $\mathcal{H}(k_1, k_2)$ respectively, and their cost by $F(k)$, $F(k_1, k_2)$, and $H(k_1, k_2)$. A flow is *locally optimal* if no other flow on the same network has lower cost. We denote by $f^*(k)$, $f^*(k_1, k_2)$, $h^*(k_1, k_2)$ locally optimal flows on $\mathcal{F}(k)$, $\mathcal{F}(k_1, k_2)$, $\mathcal{H}(k_1, k_2)$ and their cost by $F^*(k)$, $F^*(k_1, k_2)$ and $H^*(k_1, h_2)$. We say that we *compute* $f^*(k)$ (resp. $f^*(k_1, k_2)$ or $h^*(k_1, k_2)$) if we compute any locally optimal flow on $\mathcal{F}(k)$ (resp. $\mathcal{F}(k_1, k_2)$ or $\mathcal{H}(k_1, k_2)$) satisfying Hypotheses 4.2 [Placement] and 4.4 [Distance]. For a flow $f(k_1)$ on $\mathcal{F}(k_1)$ and a flow $h(k_1, k_2)$ on $\mathcal{H}(k_1, k_2)$, we define the sum $f(k_1) + h(k_1, k_2)$ as the sum of these two flows on $\mathcal{F}(k_1, k_2)$.

4.4.2 ZIO Property

For P_{ww}, the ZIO property for the dynamic lot-sizing problem gives the formula:

$$F^*(k_1, k_2) = F^*(k_1) + H^*(k_1, k_2)$$

However, the networks $\mathcal{F}(k_1)$ and $\mathcal{H}(k_1, k_2)$ overlap on $R + 2$ nodes: v_{pur}, $v_{man}(k_2+1, 0)$ and every manufacturer node $v_{man}(k_2, r)$ for $r \in [0, R[$. While v_{pur} is also common node in P_{ww}, the manufacturer nodes are not. The following proposition adapts the ZIO property to our container management problem. It states that, despite $\mathcal{F}(k_1)$ and $\mathcal{H}(k_1, k_2)$ overlapping, any pair of locally optimal solutions on $\mathcal{F}(k_1)$ and $\mathcal{H}(k_1, k_2)$ builds a locally optimal solution on $\mathcal{F}(k_1, k_2)$.

Proposition 4.4.1 *We assume that Hypotheses 2.1 [Delay], 2.2 [Cost], 4.1 [Demand], 4.2 [Placement], and 4.4 [Distance] hold. Let $k_1 \in [0, T-2]$ and $k_2 \in [k_1+2, T]$. For every pair of locally optimal flows $f^*(k_1)$ and $h^*(k_1, k_2)$, the flow defined by $f^*(k_1) + h^*(k_1, k_2)$ is locally optimal on $\mathcal{F}(k_1, k_2)$.*

To prove Proposition 4.4.1, we define new notations. We define a *task* A as the set of consecutive demand units with the same parameters r and i. For an early time step, an *early task* is written:

$$A(r, i, early) := \{D(t, r, i), t : D(t, r) \geq i\} \tag{4.2}$$

For a late time step, a *late task* is written as one of:

$$A(r, i, odd) := \{D(t, r, i), t : odd, D(t, r) \geq i\} \tag{4.3}$$

$$A(r, i, even) := \{D(t, r, i), t : even, D(t, r) \geq i\}. \tag{4.4}$$

Since the demand is non-decreasing, tasks are repeating. Early tasks repeat every period and late tasks every two periods. We denote by $L(A)$ the *periodicity* of a task A so that $L(A) := 1$ for early tasks and $L(A) := 2$ for late tasks. Given a flow f on a network, we assign tasks to containers, i.e. we tell which container fulfills which task at which period. This operation creates a *container assignment* which does not change the flow f.

Lemma 4.4.2 *For every flow, we can assign tasks to containers so that if a task A is fulfilled at periods t and $t + L(A)$. then it is fulfilled by the same container.*

Proof:

Lemma 4.4.2 is proved by induction on t using the fact that a container fulfilling task A at period t can be ordered at period $t + L(A)$.
Consider a task A which is fulfilled at periods t and $t+L(A)$. The container $Cont_1$ fulfilling task A at period t arrives to the manufacturer stock after ordering at period $t + L(A) - 1$ and before ordering at period $t + L(A)$. At period $t + L(A)$, a container $Cont_2$ is sent to the supplier to fulfill task A. If containers $Cont_1$ and $Cont_2$ are not the same, we can exchange the movements of these containers starting from the beginning of period $t+L(A)$, so that $Cont_1$ is used to fulfill task A at time $t + L(A)$ and $Cont_2$ do what $Cont_1$ was doing before exchanging the containers. The flow is exactly the same, and task A is fulfilled by the same container at times t and $t + L(A)$. At the initialization of the induction, we have $t = 0$. The exchange of containers we just described does not change what the containers were doing before time $t + L(A)$. Thus, after we exchanged containers for task A for $t = 0$, the container $Cont_1$ is busy fulfilling task A from period 0 to period $2 \cdot L(A) - 1$. Exchanging other containers at time $t = 0$ will not prevent $Cont_1$ from fulfilling task A at times $t = 0$ and $t = L(A)$. Thus, we can iterate on the tasks A at period 0 to create a container assignment so that the lemma holds for $t = 0$.
For the induction step, we assume that we have container assignment so that the lemma holds for all $t < k$, for some $k \geq 0$. Since the exchange of

containers does not change the assignment up to period $k-1$, iterating on the tasks at period k transforms the container assignment into a container assignment so that the lemma holds for all $t \leq k$.

□

Let $\mathcal{CA}[f]$ be a function generating from flow f a container assignment for which the property of Lemma 4.4.2 holds. Note that we use $\mathcal{CA}[f]$ for the sake of our proof and do not need to compute it in our algorithm.

Lemma 4.4.3 *Under Hypotheses 2.1 [Delay] and 2.2 [Cost], for any flow f on a network, $\mathcal{CA}[f]$ is so that:*

1. *a container that fulfills an early task A at period t is not idle at $t+1$,*
2. *a container that fulfills a late task A at period t is not idle at $t+3$.*

Proof:

Suppose that a container fulfills an early task $A(r, u, early)$ at period t and is idle at period $t+1$. By Lemma 4.4.2 task A is not fulfilled at $t+1$. Since the demands are profitable, the solution is not optimal and we should have used the container to fulfill task A at period $t+1$.
Suppose that a container fulfills a late task $A(r, u, odd)$ or $A(r, u, even)$ at period t and is idle at period $t+3$. During period $t+1$, this container is still fulfilling task A. It must be idle during period $t+2$, as otherwise it fulfills an early task, which is impossible because we just proved that a container cannot be idle after an early task. Thus, the container is idle during periods $t+2$ and $t+3$ and could have been used to fulfill task A at period $t+2$. Since the demands are profitable, and 4.4.2, the solution is not optimal.

□

Corollary 4.4.4 *Let f^* be a locally optimal flow on a network. Under Hypothesis 2.1 [Delay] and 2.2 [Cost], after its first task, a container in $\mathcal{CA}[f]$ alternates between fulfilling tasks and be idle for at most one period.*

Proof:

By Lemma 4.4.3, a container is idle at most one period after fulfilling a late task and is never idle after fulfilling an early task.

□

Lemma 4.4.5 *In any network, there is a locally optimal flow f^* to $\mathcal{F}(k)$ so that if k_1 is a placement, then every container in $\mathcal{CA}[f^*]$ purchased at period $k < k_1$ fulfills at least one task before period k_1.*

Proof:

Suppose that a container stays idle from its purchasing time k to placement k_1. Under stationary costs, we can reduce the cost of the flow by $(k_1 - k) \cdot$

$R \cdot C_{man}$ by purchasing the container at period k_1 instead. Under non-stationary costs, there are three possibilities:

1. If it is more profitable to purchase containers at period k and let them idle until period k_1, then the solution is not locally optimal as we should not purchase containers at period k_1.

2. If it is more profitable to purchase containers at period k_1 directly, then the solution is not locally optimal as containers purchased at time k and idle until placement k_1 incur an additional cost.

3. If both alternatives induce the same cost, then we can shift the purchasing of all containers from period k_1 to period k. If in addition $C_{setup}(k_1) > 0$, the shifting decreases the cost. Otherwise, the flow has the same cost, so we can choose a locally optimal flow where containers purchased at period k are not always idle until period $k_1 > k$.

□

Lemma 4.4.6 *Suppose that Hypotheses 2.1 [Delay], 2.2 [Cost], 4.1 [Demand], and 4.2 [Placement] hold. Let* $k_1 \in [0, T-3]$ *and* $k_2 \in [k_1+2, T[$. *Then in any locally optimal flow* $h^*(k_1, k_2)$ *on network* $\mathcal{H}(k_1, k_2)$, *arc* $a_{end}(k_2+1)$ *is empty*[1].

Proof:

We have to prove that every container in $h^*(k_1, k_2)$ is busy during period $k_2 + 1$, since the flow in $a_{end}(k_2 + 1)$ corresponds to idle containers.
We first generate the container assignment $\mathcal{CA}[h^*(k_1, k_2)]$. The containers in the system at period k_1 can be divided into two groups: The containers from the first group are purchased at period k_1, while the containers from the second group are purchased before period k_1 and is represented by the arrival of containers between time $(k_1 + 1, 1)$ and time $(k_1 + 2, L_{del} - 1)$.
By Lemma 4.4.5, containers from the first group fulfill at least one task before period k_2. In addition, containers from the second group are busy during period $k_2 + 1$. By Corollary 4.4.4, containers from both groups are not idle at period $k_2 - 1$ or k_2. By Hypothesis 4.2 [Placement], every task fulfilled at period $k_2 - 1$ or k_2 is also fulfilled at periods k_2 and $k_2 + 1$.
Therefore, in $\mathcal{CA}[h^*(k_1, k_2)]$, containers from either group are busy during periods k_2 and $k_2 + 1$. It follows that no container is in the manufacturer stoch after ordering at period $k_1 + 1$ and thus the flow in $a_{end}(k_2 + 1)$ is zero.

□

[1] Note that $k_2 \neq T$ in this lemma.

Using the same lead as in Lemma 4.4.6, we prove Proposition 4.4.1:

Proof (Proposition 4.4.1):

We want to prove that any locally optimal flow $f^*(k_1, k_2)$ in $\mathcal{F}(k_1, k_2)$ has the same cost as $F^*(k_1) + H^*(k_1, k_2)$, for any locally optimal flows $f^*(k_1)$ and $h^*(k_1, k_2)$.
We generate the container assignment $\mathcal{CA}[f^*(k_1, k_2)]$. By Lemma 4.4.5, Corollary 4.4.4, Hypotheses 4.2 [Placement] and 4.4 [Distance], we deduce with the same reasoning as for Lemma 4.4.6 that no container in $\mathcal{CA}[f^*(k_1, k_2)]$ is idle at period $k_1 + 1$. Thus, containers in $\mathcal{CA}[f^*(k_1, k_2)]$ can be divided into:

1. containers that are either:
   - purchased at a period $k < k_1$ and not idle at periods k_1 and k_1+1,
   - purchased at period k_1 and not idle at period $k_1 + 1$.
2. containers that are either:
   - purchased at period k_2,
   - purchased at period k_1, idle at periods k_1 and $k_1 + 1$, and only fulfilling tasks after period $k_1 + 2$.

The first group of containers builds a container assignment on $\mathcal{F}(k_1)$. The second group builds a container assignment on $\mathcal{H}(k_1, k_2)$. By local optimality of $f^*(k_1)$ and $h^*(k_1, k_2)$, we get: $F^*(k_1) + H^*(k_1, k_2) \leq F^*(k_1, k_2)$. Since $F^*(k_1, k_2)$ is locally optimal and $f^*(k_1) + h^*(k_1, k_2)$ is a flow on $\mathcal{F}^*(k_1, k_2)$, it follows that $F^*(k_1, k_2) = F^*(k_1) + H^*(k_1, k_2)$ and thus $f^*(k_1) + h^*(k_1, k_2)$ is locally optimal on $\mathcal{F}(k_1, k_2)$.

□

4.4.3 Polynomial Algorithm

We deduce from Proposition 4.4.1 the following algorithm adapting the W-W algorithm to the deterministic container purchasing problem.

Algorithm 2: Algorithm *Flow*.4.1

Compute $f(0)$ and $F(0)$;
for $k_2 : 1 \to T$ **do**
  $k_1 := argmin\{F(k) + H(k, k_2),\ k \in [0, k_2[\ \}$;
  $f(k_2) := f(k_1) + h(k_1, k_2)$;
return $f(T)$

Theorem 4.4.7 *We assume that the Hypotheses 2.1 [Delay], 2.2 [Cost], 4.1 [Demand], 4.2 [Placement] and 4.4 [Distance] hold. Let $\mathcal{M}cf$ be the complexity of a minimum linear-cost flow on any network $\mathcal{H}(k_1, k_2)$. Algorithm Flow*.4.1 *computes an optimal deterministic policy in $O(T^2 \cdot \mathcal{M}cf)$ time.*

Proof:

At each of the $O(T)$ iterations, we compute $O(T)$ minimum linear-cost flows on an uncapacitated network $\mathcal{H}(k_1, k_2)$ with $O(R \cdot T)$ nodes and $O(R \cdot T)$ arcs. The optimality follows from Proposition 4.4.1.

□

Remark 4.4.8 *If the cost assumption described in (2.5) does not hold, we have to consider a more complex network pattern (see Figure 3.4b in Chapter 3). The adapted pattern requires to use $R \cdot T$ more nodes and $2 \cdot R \cdot T$ more arcs, so the complexity bound is the same despite the running time most likely increasing by* 50%.

Proposition 4.4.9 *Using the enhanced capacity scaling algorithm described by Ahuja et al. [2] and the Dijkstra algorithm to compute shortest paths, Algorithm Flow.*4.1 *runs in $O(R^2 \cdot T^4 \cdot \log[R \cdot T]^2)$ time.*

We conclude this section with a proposition on the structure of the locally optimal flows.

Remark 4.4.10 *Let $0 \leq k_1 < k_2 \leq T$ and $h^*(k_1, k_2)$ be a locally optimal flow. If $k_2 = T$, then no container is purchased at period k_2, since we are past the end of the time horizon. otherwise, the flow in $a_{end}(k_2+1)$ is empty by Lemma 4.4.6. Thus, v_{pur} is incident to at most two arcs with positive flow.*

4.5 Solution Allowing Close Placements

We now relax Hypothesis 4.4 [Distance], namely that:

1. period 0 must be a placement,
2. period $T-1$ should not be a placement,
3. consecutive periods cannot be simultaneously placements.

Firstly, relaxing the constraint at period 0 is straightforward. We only need to create dummy periods with zero-demand before period 0, such that the algorithm can force the first period to be a placement without loss of generality. Since we will allow consecutive periods to be simultaneously placements, we only need to add a single dummy period -1. Otherwise, we would have to add two dummy periods -1 and -2. On the dummy period -1, there is no demand and the manufacturer holding cost is infinite so that no container can be in stock at the beginning of period 0.
Secondly, consider the assumption that period $T-1$ cannot be a placement. This assumption is needed for Algorithm *Flow*.4.1 because it requires period

T to be a placement in order for $f^*(T)$ to be the optimal solution. We relax it by adding a dummy period $T+2$ without demand or cost, and by returning $f^*(T+1)$ instead of $f^*(T)$.
Finally, we adapt our algorithm to allow close placements. For that purpose, we extend the networks functions $\mathcal{F}$ and $\mathcal{H}$, and along with them the notations f, f^*, F, F^*, h, h^*, H and H^*. Our generalizations come with the following new definition:

Definition 4.5.1 *We call a placement k* grouped *if either periods $k-1$ or $k+1$ is also a placement, and* isolated *otherwise. For $-1 \leq k_1 \leq k_2 \leq T$, we call interval $[k_1, k_2]$ a* placement interval *if $k_1 - 1$, $k_2 + 1$ are not placements and every period $k \in [k_1, k_2]$ is a placement. A placement k is called* odd *if it is at an odd position in its placement interval, and* even *otherwise. In particular, in the placement interval $[k_1, k_2]$, the first placement k_1 is odd.*

For the extension of network $\mathcal{H}$, we use the same structure as previously so that each network still contains at least four periods. However, we consider four possible patterns of the network describing the different possibilities, whether periods $k_1 + 1$ and $k_2 + 1$ are placements or not.
Consider a placement $k_1 \in [-1, T-2]$ and the first placement $k_2 \in [k_1+2, T]$ at least two periods later. We replace network $\mathcal{H}(k_1, k_2)$ with four networks $\mathcal{H}_{i,j}(k_1, k_2)$ with $i, j \in \{1, 2\}$ so that (see Figure 4.6):

- periods k_1 and k_2 are placements.
- period $k_1 + 1$ is a placement if and only if $i = 2$.
- period $k_2 + 1$ is a placement if and only if $j = 2$.
- if $i = 2$ and $k_1 + 2 < k_2$, then we remove the arcs $a_{dis}(k_1 + 2, r)$ for $r \in [0, R[$, so that no disposable is allowed at period $k_1 + 2$.
- placements k_1 and k_2 are odd.

Consequently, for $k_1 \geq 0$, networks $\mathcal{H}_{1,1}(k_1, k_2)$ and $\mathcal{H}(k_1, k_2)$ are identical.

Remark 4.5.2 *The last property of the definition of $\mathcal{H}_{i,j}(k_1, k_2)$ is very important for our optimality proof. However, because of it we need to add the dummy period $T + 2$ and return the best policy up to period $T + 1$.*

For $k \in [-1, T]$, and $j \in \{1, 2\}$, we define the networks $\mathcal{F}_j(k)$ so that:

- The only demand nodes are $v_{sup}(t, r)$ with $r \in [0, R[$ and $t \in [-1, k+2]$.
- The only manufacturer nodes are $v_{man}(t, r)$ with $(t, r) \leq (k + 1, 0)$.
- Period k is an odd placement.
- Period $k + 1$ is a placement if and only if $j = 2$.

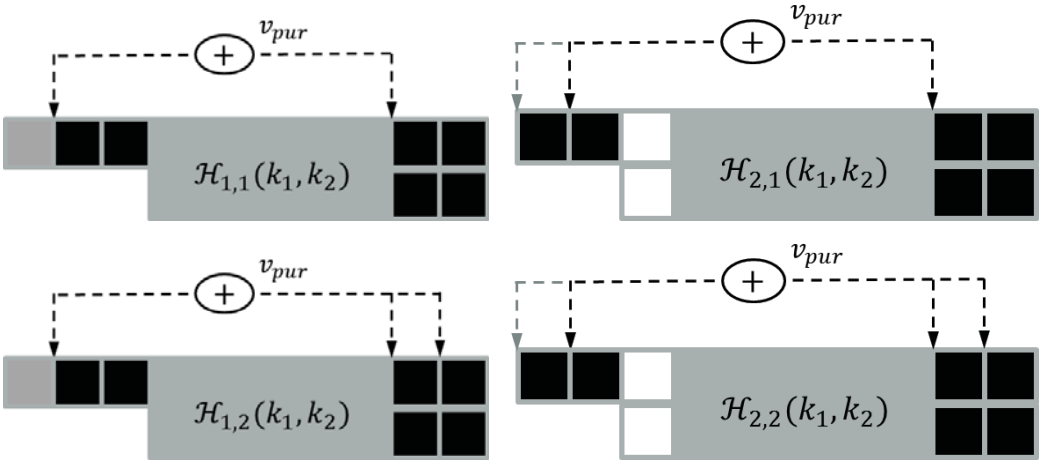


Figure 4.6: Four networks $\mathcal{H}_{i,j}$ to compute in the algorithm. The black rectangles represent periods k_1, k_1+1, k_2 and k_2+1. When $j=2$, the white rectangles represent period k_1+2, where the demand has not been fulfilled yet, must must be entirely fulfilled.

For $k_1 \in [-1, T-2]$, $k_2 \in [k_1+2, T]$, $i \in \{1,2\}$ and $j \in \{1,2\}$, we define networks $\mathcal{F}_{i,j}(k_1,k_2)$ from $\mathcal{F}(k_1,k_2)$ so that:

- k_1 and k_2 are odd placements.
- k_1+1 (resp. k_2+1) is a placement if and only if $i=2$ (resp. $j=2$).
- no period on $[k_1+2, k_2-1]$ is a placement.

Lemma 4.5.3 *We have, for all $k \in [0,T]$:*

$$F_j(k) = \min\left\{F_{i,j}(k_1,k);\ k_1 \in [0,k-2],\ i \in \{1,2\}\right\} \tag{4.5}$$

Proof:

Take a locally optimal solution $f_j^*(k)$ satisfying Hypothesis 4.2 [Placement]. We recall that we set our first placement at the dummy period -1, and the manufacturer holding stock at this period is infinite. Therefore, if k is the first placement starting from $t \geq 0$ in the policy $f_j^*(k)$, then:

$$f_j^*(k) = f_{1,j}(0,k)$$

and placement k is odd due to being the first of its interval. Otherwise, let k_1 be the last placement before k.

- If $k_1 < k-1$ and k_1 is odd, then placement k is odd due to being the first element of its placement interval and we have:

$$f_j^*(k) = f_{1,j}(k_1,k)$$

- If $k_1 < k-1$ is even, then period $k_1 - 1$ must be odd. Placement k is also odd as the first element of its interval and:

$$f_j^*(k) = f_{2,j}(k_1 - 1, k)$$

- If $k_1 = k-1$ is even, then periods k and k_1 belong to the same interval and placement k is odd. Moreover, placement k_1 being even means that period $k_1 - 1$ is an odd placement, so the policy $f_j(k)$ can be written as:

$$f_j^*(k) = f_{2,j}(k_1 - 1, k)$$

- If $k_1 = k-1$ is odd, then periods k and k_1 belong to the same interval. However period k must be odd, which contradicts our assumption. Even though this can occur in a locally optimal policy, it does not correspond to a value $f_j(k)$, so this case is disregarded.

□

Remark 4.5.4 *Since there is no new demand after period $T-1$, we can assume without loss of generality that the optimal policy to compute does not purchase any container at period T.*

Corollary 4.5.5 *Both policies $f_1^*(T+1)$ and $f_2^*(T+1)$ are optimal solutions to the deterministic container purchasing problem under Hypotheses 2.1 [Delay], 2.2 [Cost], 4.1 [Demand] and 4.2 [Placement].*

Since Lemmas 4.4.2, 4.4.3, 4.4.5 and Corollary 4.4.4 do not require Hypothesis 4.4 [Distance], they also hold in this section. Therefore, we only have to extend Proposition 4.4.1.

Proposition 4.5.6 *Let $i, j \in [1,2]$, $k_1 \in [-1, T-2]$ and $k_2 \in [k_1+2, T]$. For any locally optimal flows $f_{i,j}^*(k_1,k_2)$, $f_i^*(k_1)$, $h_{i,j}^*(k_1,k_2)$, we have that:*

$$F_{i,j}^*(k_1,k_2) = F_i^*(k_1) + H_{i,j}^*(k_1,k_2) \tag{4.6}$$

Proof:

This proposition follows the same lead as Proposition 4.4.1. We consider a container assignment $\mathcal{CA}$ to $f_{i,j}^*(k_1,k_2)$ as explained in Lemma 4.4.2. Under $\mathcal{CA}$ and by Hypothesis 4.1 [Demand], every container purchased at a placement $k < k_1$ will be busy fulfilling a task during cycles k_1 and k_1+1. Therefore, containers are divided into two groups:

1. Containers purchased up to k_1+1 and fulfilling some tasks during period k_1+1, building a flow on $\mathcal{F}_i(k_1)$.

2. Containers purchased starting from k_1 and idle up to period k_1+2, building a flow on $\mathcal{H}_{i,j}(k_1,k_2)$.

Thus, $f^*_{i,j}(k_1, k_2)$ is composed of one flow on $\mathcal{F}_i(k_1)$ and one on $\mathcal{H}_{i,j}(k_1, k_2)$ and the proposition holds.

□

Algorithm 3: Algorithm *Flow*.4.2

Compute $f(0)$ and $F(0)$;
for $k_2 : 1 \to T+1$ **do**
  $k_{1,1} := \text{argmin}\{F_1(k) + H_{1,1}(k, k_2), k \in [-1, k_2 - 1]\}$;
  $k_{2,1} := \text{argmin}\{F_2(k) + H_{2,1}(k, k_2), k \in [-1, k_2 - 1]\}$;
  $k_{1,2} := \text{argmin}\{F_1(k) + H_{1,2}(k, k_2), k \in [-1, k_2 - 1]\}$;
  $k_{2,2} := \text{argmin}\{F_2(k) + H_{2,2}(k, k_2), k \in [-1, k_2 - 1]\}$;
  $f^*_1(k_2) :=$
   $\text{argmin}\{F^*_1(k_{1,1}) + H^*_{1,1}(k_{1,1}, k_2), F^*_2(k_{2,1}) + H^*_{2,1}(k_{2,1}, k_2)\}$;
  $f^*_1(k_2) :=$
   $\text{argmin}\{F^*_1(k_{1,2}) + H^*_{1,2}(k_{1,2}, k_2), F^*_2(k_{2,2}) + H^*_{2,2}(k_{2,2}, k_2)\}$;
return $F_1(T+1)$

Theorem 4.5.7 *Let $\mathcal{M}cf$ be the complexity of computing a minimum linear-cost flow on a network $\mathcal{H}_{i,j}(k_1, k_2)$. If Hypotheses 2.1 [Delay], 2.2 [Cost], 4.1 [Demand] and 4.2 [Placement] hold, Algorithm Flow*.4.2 *computes an optimal policy in $O(T^2 \cdot \mathcal{M}cf)$ time.*

Proposition 4.5.8 *Using the Dijkstra algorithm to compute shortest paths and the enhanced capacity scaling algorithm to compute minimum linear-cost flows, Algorithm Flow*.4.2 *takes $O(R^2 \cdot T^4 \cdot \log[R \cdot T]^2)$ time.*

We conclude with a last small result:

Lemma 4.5.9 *For $j \in \{1, 2\}$, there is a locally optimal flow $h^*_{2,j}(k_1, k_2)$ on $\mathcal{H}_{2,j}(k_1, k_2)$ with an empty flow in $a_{pur}(k_1)$.*

Proof:

Since a flow $h_{2,j}(k_1, k_2)$ on $\mathcal{H}_{2,j}(k_1, k_2)$ only satisfies demands starting from period $k_1 + 2$, a container purchased at period k_1 in $h_{2,j}(k_1, k_2)$ will necessarily stay idle during periods k_1 and $k_1 + 1$, while purchasing at period $k_1 + 1$ only requires the containers to be idle at period $k_1 + 1$.
Therefore, unless the container cost $C_{cont}(t)$ is much cheaper for $t = k_1$ and $t = k_2$, purchasing containers at period k_1 incurs a higher cost. Otherwise, it is not profitable to have a placement at period $k_1 + 1$ if we have one at period k_1. Thus, because of the positive setup cost $C_{setup}(k_1+1)$, $h^*_{2,j}(k_1, k_2)$ does not belong to any optimal solution to the problem.

□

4.6 Extensions

In this section, we consider several extensions of either of the two algorithms we described. For the sake of simplicity, we assume stationary costs and describe the extensions for Algorithm *Flow*.4.1.

4.6.1 Several Suppliers

We suppose now that we have J suppliers numbered $1, \ldots, J$. For Supplier $j \in [1, n]$, we denote then by $L_{del}[j] < R$ the delivery delay to the manufacturer, by $C_{sup}[j]$ the holding cost, by $C_{dis}[j]$ the disposable cost and by $D[j](t, r)$ the demand at time (t, r). The ordering delay is still assumed zero for each supplier, without loss of generality.
The suppliers are not required to order at the same time, but have the same ordering periodicity of R time steps. We denote by $\rho[j] \in [0, R[$ the ordering step of supplier j. The manufacturer can only purchase containers at the beginning of a period.

Lemma 4.6.1 *The updated network for J suppliers has $R \cdot J \cdot (T+2) + R \cdot (T+1) + 2$ nodes and $(T+1) \cdot \big((J+2) \cdot R + 1\big) + 1$ arcs.*

We generalize Hypothesis 4.1 [Demand] as following:

Hypothesis 4.5 ***[Demand-Sup]:*** *For each $r \in [0, R[$ and $j \in [1, J]$, the demand $D(t, r)[j]$ is non-decreasing in t.*

Since the manufacturer is purchasing, Lemmas 4.4.2, 4.4.3, 4.4.5 as well as Corollary 4.4.4 can be trivially generalized to J suppliers. Thus, with exactly the same reasoning, we can generalize Proposition 4.4.1. Therefore, our algorithm also compute an optimal solution in a setting with J suppliers, by updating the networks $\mathcal{H}$ and $\mathcal{F}$ so that every supplier gets the same treatment as in the single supplier case.
We conclude:

Theorem 4.6.2 *Under Hypotheses 4.2 [Placement] and 4.5 [Demand-Sup], Algorithm Flow*.4.1 *can be extended to J suppliers to solve DCPP optimally in $O(T^4 \cdot R^2 \cdot J^2 \cdot \log[T \cdot R \cdot J]^2)$ time.*

4.6.2 Longer Lead Times

We suppose that the delivery delay can take any non-infinite integral value.

Hypothesis 4.6 ***[GenDelay]:*** *There is a constant $\omega > 0$ so that:*

$$(\omega - 1) \cdot R < L_{del} \leq \omega \cdot R \tag{4.7}$$

Hypothesis 2.1 [Delay] corresponds to the special case $\omega = 1$. We had early tasks with periodicity of one period and late tasks with periodicity of two periods. For $\omega \geq 1$, we still have two kind of tasks, namely early tasks with periodicity of ω periods and late tasks with periodicity of $\omega + 1$ periods. We update Hypothesis 4.2 [Placement] to:

Hypothesis 4.7 ***[Placement-GenDelay]*** *There is an optimal solution to DCPP so that if $k \in [0, T]$ is a placement, then no disposable is bought from period k to $k + \omega$.*

Lemmas 4.4.2 still holds because its statement is generic in the periodicity of tasks. Lemma 4.4.5 also holds. Lemma 4.4.3 must be generalized as:

Lemma 4.6.3 *Suppose that Hypotheses 2.2 [Cost], 4.6 [GenDelay], and 4.7 [Placement-GenDelay] hold. For any flow f, $\mathcal{CA}[f]$ is such that a container fulfilling a task A of length $L(A)$ is not idle in period $t + 2 \cdot L(A) - 1$.*

Corollary 4.4.4 becomes:

Corollary 4.6.4 *Let f be a locally optimal flow on a network. Then, after its first task, a container in $\mathcal{CA}[f]$ alternates between fulfilling tasks and be idle for at most ω periods.*

A generalization of Proposition 4.4.1 follows with updated networks $\mathcal{F}$ and $\mathcal{H}$ so that we have $\omega + 1$ special periods after each placement instead of two. We update Hypothesis 4.4 [Distance] to:

Hypothesis 4.8 ***[Distance-GenDelay]:*** *There is an optimal solution fulfilling Hypothesis 4.7 [Placement-GenDelay] so that:*

- *period 0 is a placement.*
- *if $k \in [0, T - 1]$ is a placement, then there is no placement from period $k + 1$ to $k + \omega$.*
- *Periods $T - \omega$ to $T - 1$ are not placements.*

Theorem 4.6.5 *We suppose that Hypotheses 2.2 [Cost], 4.6 [GenDelay], 4.7 [Placement-GenDelay] and 4.8 [Distance-GenDelay] hold. Then, an algorithm similar to Algorithm Flow.4.1 solves optimally DCPP in $O(R^2 \cdot T^4 \cdot \omega^2 \log[R \cdot T]^2)$ time.*

Proof:

The algorithm is straightforward to generalize and is correct by the above analysis. The factor ω^2 is added to the complexity because there is at most one placement every ω periods.

□

However, when Hypothesis 4.8 [Distance-GenDelay] does not hold, a direct extension of Algorithm *Flow*.4.2 has to consider every placement possibilities for the $\omega+1$ consecutive periods following a placement, as we presented in Figure 4.6 in the case $\omega = 1$. Therefore, a direct generalization gives a algorithm polynomial in T and R but exponential in ω.

Theorem 4.6.6 *Suppose that Hypotheses 2.2 [Cost], 4.6 [GenDelay] and 4.7 [Placement-GenDelay] hold. We can update Algorithm Flow*.4.2 *to solve DCPP optimally in* $O(R^2 \cdot T^4 \cdot 4^\omega \log[R \cdot T]^2)$ *time.*

4.6.3 Bi-Objective Optimization

In our modeling, we made the simple assumption that the placements can be modeled by a setup cost like in most lot-sizing problems. An alternative modeling is to consider the number of placements as a different objective function.

We compute the minimum cost policy using $K \in [1, T]$ placements. Afterward, a manager decides on the best compromise between cost and number of placements.

Instead of computing the locally optimal solutions $f^*(k)$ up to period $k+1$, we add a parameter $n \geq 2$ and compute the locally optimal solutions $f^*(k, n)$ containing exactly n placements before period k. Therefore, we consider the same costs $H^*(k_1, k_2)$ and update the dynamic programming search such that:

$$\forall k \in [1, T[: \;\; f^*(k, 1) := h^*(0, k) \tag{4.8}$$

$$\forall k_2 \in [1, T[, \forall n \geq 1 : \;\; F^*(k_2, n+1) := \min_{k_1 < k_2} \left[F^*(k_1, n) + H^*(k_1, k)\right] \tag{4.9}$$

The time complexity of this dynamic program is thus $O(T^3)$, as we iterate over $k_2 \in [1, T]$, $k_1 \in [0, k_2[$ and $n \in [1, T[$.

Finally, we define the *pareto optimal solutions* so that no other solution is better, i.e. every solution either induces a greater cost or a different number of placements. The solution of this bi-objective problem is the set of $O(T)$ pareto optimal solutions, that is one minimum cost policy for each number of placements.

We point out that a minimum linear-cost flow on a network $\mathcal{H}(k_1, k_2)$ takes at least $O(R \cdot T)$ time, as we need to assign a flow to each of the $O(R \cdot T)$ sinks. We conclude:

Theorem 4.6.7 *The bi-objective problem can be solved in* $O(T^2 \cdot \mathcal{M}cf)$ *time, where* $\mathcal{M}cf$ *is the cost of a minimum linear-cost flow on a network* $\mathcal{H}(k_1, k_2)$.

4.6.4 General Demand Patterns

We now look at the behavior of Algorithm $Flow$.4.1 under a general demand pattern. Consider two values k_1 and k_2 so that $0 \leq k_1 < k_2 \leq T$, and network $\mathcal{H}(k_1, k_2)$. In the case of a non-decreasing demand pattern, we have shown with Lemma 4.4.6 that arc $a_{end}(k_2+1)$ is empty in any locally optimal flow. However, when the demand follows a general demand pattern, it may be optimal to purchase more containers at period k_1 and let them idle during periods k_2 and $k_2 + 1$. This occurs in particular when the purchasing costs $C_{setup}(k_2)$ and $C_{cont}(k_2)$ as well as the demand at periods k_2 and $k_2 + 1$ are very low. It is then more profitable to purchase new containers at period k_2, but the total demand at this period will be lower than the optimal fleet size to have before period k_2.

If $k_2 < T$, our algorithm does not compute a feasible solution, because the arc $a_{end}(k_2 + 1)$ is only allowed to be empty when $k_2 = T$. Indeed, the algorithm will purchase the additional profitable containers using arc $a_{pur}(k_1)$ and send the resulting flow back to v_{pur} using arc $a_{end}(k_2 + 1)$. The flow will not be a feasible solution because we have currently no way of transposing the flow in $a_{end}(k_2 + 1)$ into network $\mathcal{G}$, unless $k_2 = T$.

We can avoid this problem by giving a very high cost to arc $a_{end}(k_2 + 1)$ so that it will never be in any locally optimal solution. In other words, we keep the property that no container in the system is idle at period $k_2 + 1$ in network $\mathcal{H}(k_1, k_2)$, as every container either fulfills a late demand of period k_2 or a demand of period $k_2 + 1$.

However, even if $a_{end}(k_2 + 1)$ has an infinite cost, it may still have a positive flow in a locally optimal solution. This occurs in particular if the total early demand at period k_2 is greater than the total demand at period $k_2 + 1$, or if the total demand at periods k_1 and $k_1 + 1$ is greater than the total demand at periods k_2 and $k_2 + 1$. The resulting over excess will necessarily be sent to other demand nodes via arc $a_{end}(k_2 + 1)$ This locally optimal flow will have a very high cost, thus the extended algorithm will choose other locally optimal flows to build a policy. We note that the locally optimal flow on network $\mathcal{H}(0, T)$ cannot have this issue, so there is least one feasible flow $h^*(0, T)$ on $\mathcal{G}$ without any flow in infinite cost arcs.

Finally, the algorithm always compute a feasible solution, but there is no guarantee that the policy is optimal, even under Hypotheses 2.1 [Delay], 2.2 [Cost], and 4.2 [Placement].

4.7 Outlook

In this chapter, we have solve optimally the deterministic container management problem under the assumptions that:

1. no disposable is bought for the two periods following a placement,

2. the demand at each time step is non-decreasing,

3. every demand is profitable.

When a demand is not profitable, removing it from the system may violate the non-decreasingness behavior of the demand. If not, the third assumption can be made without loss of generality.
We have presented three algorithms. Firstly, the problem without disposable can be reduced to the dynamic lot-sizing problem, making it easy to solve. The two other algorithms adapts the W-W algorithm to our network by computing minimum linear-cost flows at every iteration. They both run in $O(R^2 \cdot T^4 \cdot \log[R \cdot T]^2)$. The first of these two algorithms is simpler to understand, but requires an additional hypothesis, whereas the second extends the first with more complex notations. This last algorithm is the main result of this chapter.
We have considered some possible extensions of these algorithm such as longer lead times and a supply chain between several suppliers and a single manufacturer.
Finally, we have updated the algorithms to generate a feasible policy for a general demand pattern, but at the cost of the solution optimality. The next chapter covers algorithms under a general demand behavior.

Chapter 5

Algorithms for General Demand

In this chapter, we solve the $DCPP$ under any demand pattern and adapt the algorithms from Chapter 4 to this setting. Sections 5.1 and 5.2 solve the problem respectively without transportation delay and without disposables. Sections 5.3 and 5.4 adapt the flow-based algorithms $Flow$.4.1 and $Flow$.4.2 to any demand pattern, under an hypothesis similar to 4.2 [Placement]. In Section 5.5, we compare the algorithms of the two chapters and present possible extensions of the new algorithms. In Section 5.6 we describe a flow decomposition using extreme points and deduce sufficient conditions to add to the system for $DCPP$ to be polynomially solvable. We conclude this chapter with an outlook.

5.1 Solution without Delay

In this section, we solve the problem under zero delivery delay. Every container used for a demand $D(t,r)$ can already be ordered at period $t+1$.

Hypothesis 5.1 ***[NoDelay]:*** *The delivery takes one time step:* $L_{del} = 1$.

Every demand is early under Hypothesis 5.1 [NoDelay]. Given a policy, we denote by β_t the order size at period t and by u_t^+ the container fleet size after possible purchasing. Then:

Lemma 5.1.1 *Under Hypotheses 2.2 [Cost] and 5.1 [NoDelay], we have for each minimum cost policy:*

$$\beta_t = \min\left\{u_t,\ \sum_{r=0}^{R-1} D(t,r)\right\} \tag{5.1}$$

Thus, no container is idle during period t *when the supplier buys disposables.*

We denote by $D(t)$ the total demand at period t. We sort the demand units in $D(t)$ by decreasing profitability such that it is always more profitable to fulfill the i-th demand unit than the $(i+1)$-th. Since every unit $D(t,r,i)$ of demand $D(t,r)$ has the same profitability, the demand units in $D(t)$ are grouped by time step. For each $r \in [0,R[$, we sort the demand units $D(t,r,i)$ in $D(t)$ by increasing unit i. We have then:

$$\begin{aligned} D(t) := \Big[& D(t,0,0), D(t,0,1), \ldots, D(t,0,D(t,0)-1), \\ & D(t,1,0), D(t,1,1), \ldots, D(t,1,D(t,1)-1), \\ & \vdots \\ & D(t,R-1,0), D(t,R-1,1), \ldots, D(t,R-1,D(t,R-1)-1) \Big] \end{aligned}$$

Remark 5.1.2 *By Hypothesis 2.2 [Cost], every demand $D(t,r)$ is more profitable than the demand $D(t,r+1)$, for $r \in [0,R-2]$. Therefore, the demand units in $D(t)$ are sorted by increasing time step. The results of this section can be generalized to the case where Hypothesis 2.2 [Cost] does not hold. We must then consider a specific sequence $r_0, \ldots r_{R-1}$ of time steps so that $D(t,r_i)$ is not less profitable than $D(t,r_i)+1$, for $r_i \in [0,R-2]$.*

For $t \in [0,T[$ and $i \in \mathbb{N}$, we define $\mathbb{G}(t,i)$ as the profitability of the i-th demand unit in $D(t)$ if $i < D(t)$ and as $-C_{idle}(t) < 0$ if $i \geq D(t)$. The negative profitability $-C_{idle}(t)$ is equal to the cost of holding a container in the manufacturer stock. Given periods k_1 and $k_2 > k_1$ We denote by $\mathbb{G}[k_1,k_2,i]$ the total profitability of fulfilling the i-th demand unit at every period of the interval $[k_1,k_2-1]$:

$$\mathbb{G}[k_1,k_2,i] := \sum_{t=k_1}^{k_2-1} \mathbb{G}(t,i) \tag{5.2}$$

By definition, we have:

Lemma 5.1.3 *If Hypotheses 2.2 [Cost] and 5.1 [NoDelay] hold, then for all $0 \leq k_1 < k_2 \leq T$, the total profitability $\mathbb{G}[k_1,k_2,i]$ is non-increasing in i.*

We denote by $h(k_1,k_2)$ a policy between periods k_1 and k_2-1 so that k_1 is the only placement, and by $H(k_1,k_2)$ its cost. We denote by $f(k)$ a policy between periods 0 and $k-1$, and by $F(k)$ its cost. Moreover, we use the notations $h^*(k_1,k_2)$, $H^*(k_1,k_2)$, $f^*(k)$ and $F^*(k)$ for locally optimal policies.

Proposition 5.1.4 *Suppose that Hypotheses 2.2 [Cost] and 5.1 [NoDelay] hold. Then we can assume without loss of generality that there is no container in the system before purchasing at period k_1 when computing $h^*(k_1,k_2)$. Therefore, we have the dynamic programming formula:*

$$\forall k_2 \in [1,T]: \; F^*(k_2) = \min_{0 \leq k_1 < k_w} \Big[F^*(k_1) + H^*(k_1,k_2) \Big] \tag{5.3}$$

Proof:

Consider an optimal policy and let k_1 and $k_2 > k_1$ be two consecutive placements. At period k_1, there are u_{k_1} containers in the system before purchasing and $u^{\alpha}_{k_1}$ afterward. If we exclude the setup cost, the gain of purchasing up to $u > u_{k_1}$ containers at period k_1 is:

$$\sum_{i=u_{k_1}+1}^{u} \big(\mathbb{G}[k_1, k_2, i] - C_{cont}(k_1)\big)$$

By Lemma 5.1.3, we have:

$$\forall i: \ \mathbb{G}[k_1, k_2, i+1] \leq \mathbb{G}[k_1, k_2, i]$$

Therefore the optimal fleet size $u^{\alpha}_{k_1}$ is such that:

$$\mathbb{G}[k_1, k_2, u^{\alpha}_{k_1} + 1] < C_{cont}(k_1) < \mathbb{G}[k_1, k_2, u_{k_1}]$$

Consequently, $u^{\alpha}_{k_1}$ does not precisely depend on the value of $u_{k_1} < u^{\alpha}_{k_1}$. If the process started without any container at period k_1, it would be the most profitable to purchase $u^{\alpha}_{k_1}$ containers.

□

We separate the possible container fleet sizes into intervals. We define $U[t, r]$ as the unit index in $D(t)$ relative to the last demand unit of $D(t, r)$, i.e. $D\big(t, r, D(t, r) - 1\big)$. We define $\mathcal{U}(k_1, k_2)$ as (see Figure 5.1):

$$\mathcal{U}(k_1, k_2) := \big\{\ U[t, r],\ t \in [k_1, k_2 - 1],\ r \in [0, R[\ \big\} \cup \{0\}$$

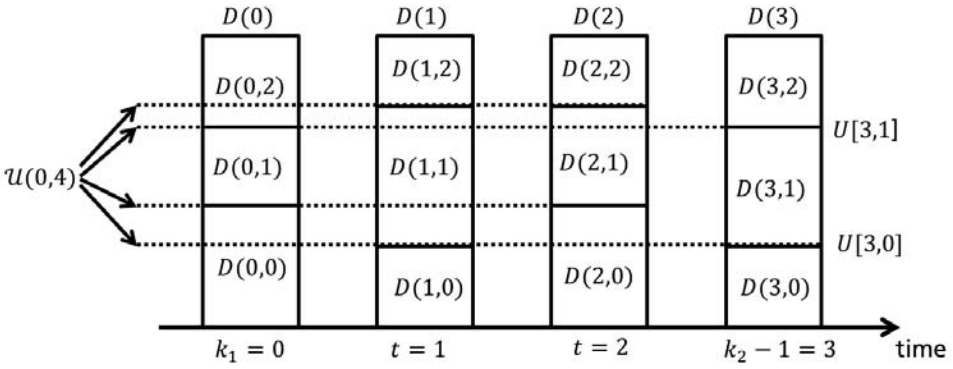


Figure 5.1: Illustration of $\mathcal{U}(0, 4)$, the possible container fleet sizes between periods $k_1 = 0$ and $k_2 - 1 = 3$.

Proposition 5.1.5 *Suppose that Hypotheses 2.2 [Cost] and 5.1 [NoDelay] hold. Consider* $0 \leq k_1 < k_2 < T$ *as well as two fleet sizes* u_1 *and* $u_2 > u_1$ *in* $\mathcal{U}(k_1, k_2)$ *such that:*

$$\forall u \in [u_1 + 1, u_2 - 1]: \ u \notin \mathcal{U}(k_1, k_2) \tag{5.4}$$

We have then:

$$\forall u \in [u_1 + 1, u_2] : \ \mathbb{G}[k_1, k_2, u] = \mathbb{G}[k_1, k_2, u_2] \tag{5.5}$$

Proof:

By definition, the demand units in $D(t)$ are grouped by time step. Let Suppose first that u_1 and $u_2 - 1$ are units from the same demand $D(t, r)$ for every $t \in [k_1, k_2[$. Then, every unit $u \in [u_1, u_2[$ also corresponds to demand $D(t, r)$ and Equation (5.5) holds. Suppose now that u_1 and $u_2 - 1$ are from different demands $D(t, r)$ and $D(t, r')$ for some $t \in [k_1, k_2[$. Then, there is a $i \in [u_1 + 1, u_2 - 1]$ corresponding to the demand unit $D\big(t, r, D(t, r) - 1\big)$. This unit i belongs to the set $\mathcal{U}(k_1, k_2)$ but unit $i + 1$ does not, so Equation (5.4) does not hold.

□

Corollary 5.1.6 *Under Hypotheses 2.2 [Cost] and 5.1 [NoDelay], there is a locally optimal policy $h^*(k_1, k_2)$ so that the container fleet size after purchasing at period k_1 is an element of $\mathcal{U}(k_1, k_2)$.*

We denote by $u^*(k_1, k_2)$ the fleet size after purchasing at period k_1 in the optimal policy described by Corollary 5.1.6:

$$u^*(k_1, k_2) := \max\{i \in \mathcal{U}(k_1, k_2), \ \mathbb{G}[k_1, k_2, i] \geq C_{cont}(k_1)\} \tag{5.6}$$

By Lemma 5.1.3, the profitability $\mathbb{G}[k_1, k_2, i]$ is non-decreasing in i, so Equation (5.6) is equivalent to the minimization problem:

$$u^*(k_1, k_2) := \arg\min \big\{u \cdot C_{cont}(k_1) - \sum_{i=0}^{u} \mathbb{G}[k_1, k_2, i] : \ u \in \mathcal{U}(k_1, k_2)\big\} \tag{5.7}$$

We define:

$$\mathbb{G}_{\sum}(t, u) := \sum_{i=0}^{u} \mathbb{G}(t, i) \tag{5.8}$$

$$\mathbb{G}_{\sum}[k_1, k_2, u] := \sum_{i=0}^{u} \mathbb{G}[k_1, k_2, i] \tag{5.9}$$

The corresponding minimum cost $H^*(k_1, k_2)$ is then:

$$\begin{aligned} H^*(k_1, k_2) = {} & C_{setup}(k_1) + u^*(k_1, k_2) \cdot C_{cont}(k_1) \\ & + \sum_{t=k_1}^{k_2-1} \sum_{r=0}^{R-1} C_{dis}(t, r) \cdot D(t, r) - \mathbb{G}_{\sum}\big[k_1, k_2, u^*(k_1, k_2)\big] \end{aligned} \tag{5.10}$$

Note that the function $\mathbb{G}_{\sum}$ represents a gain instead of a cost. Algorithm 4 uses a dynamic program similar to the W-W algorithm to compute the best policy from the values $H^*(k_1, k_2)$.

Algorithm 4: Algorithm *NDel.D*

Sequence the demand units by decreasing profitability;
// Compute $\mathcal{U}(0,T)$
$\mathcal{U}(0,T) = \emptyset$;
for $t : 0 \to T-1$ **do**
  $\mathcal{U}(t,t) = \emptyset$;
  for $r : 0 \to R-1$ **do**
    Retrieve $U[t,r]$ from $D(t)$;
    $\mathcal{U}(t,t+1) = \mathcal{U}(t,t+1) \cup U[t,r]$;
  $\mathcal{U}(0,T) = \mathcal{U}(0,T) \cup \mathcal{U}(t,t+1)$;
  Sort$(\mathcal{U}(t,t))$;
Sort$(\mathcal{U}(0,T))$;
// Compute $\mathbb{G}_{\sum}[k_1,k_2,u]$
foreach $u \in \mathcal{U}(0,T)$ **do**
  for $t : 0 \to T-1$ **do**
    Compute $\mathbb{G}_{\sum}(t,u)$;
  for $k_1 : 0 \to T-1$ **do**
    $\mathbb{G}_{\sum}[k_1,k_1+1,u] := \mathbb{G}_{\sum}(k_1,u)$;
    for $k_2 : k_1+2 \to T$ **do**
      $\mathbb{G}_{\sum}[k_1,k_2,u] = \mathbb{G}_{\sum}[k_1,k_2-1,u] + \mathbb{G}_{\sum}(k_2-1,u)$;
// Compute u^*
for $k_1 : 0 \to T-1$ **do**
  $\mathcal{U} = \emptyset$;
  for $k_2 : k_1 \to T$ **do**
    $\mathcal{U} = \mathcal{U} \cup \mathcal{U}(k_2,k_2+1)$;
    $u^*(k_1,k_2) = \arg\max\left\{\mathbb{G}_{\sum}[k_1,k_2,u] - u \cdot C_{cont}(k_1) : \ u \in \mathcal{U}\right\}$;
    Deduce $H^*(k_1,k_2)$ using Equation (5.10);
// Compute the solution
$F^*(0) = 0$;
for $k_2 : 1 \to T$ **do**
  $k_1 := \arg\min\left\{F^*(k) + H^*(k,k_2);\ k \in [0,k_2[\right\}$;
  $f^*(k_2) := f^*(k_1) + h^*(k_1,k_2)$;
return $f^*(T)$;

Proposition 5.1.7 *If Hypotheses 2.2 [Cost] and 5.1 [NoDelay] holds, then Algorithm NDel.D computes an optimal solution in* $O(R \cdot T^3)$ *time.*

Proof:

By Corollary 5.1.6, the algorithm computes the exact values for $H^*(k_1,k_2)$. Therefore, the W-W dynamic program computes an optimal solution under

the two hypotheses, by Proposition 5.1.4.
We now analyze the time complexity. The algorithm is divided into five parts. In the first part, we genenrate for each period t the demand $D(t)$. The profitability of each demand unit is implicitely computed from the index $U[t,r]$. The first part hence takes $O(R \cdot T)$ time.
In the second part, we compute the set $\mathcal{U}(0,T)$ of every possible fleet size possibly corresponding to a locally optimal policy $h^*(k_1,k_2)$. There are at most $R \cdot T$ possible values of the set of all $U[t,r]$, hence possible optimal fleet sizes. It takes $(R \cdot T)$ time to generate the set $\mathcal{U}(0,T)$ and $O\big(R \cdot T \cdot \log[R \cdot T]\big)$ time to sort it.
In the third part, we compute the profitabilities. We compute the single period profitabilities $\mathcal{U}(t,t+1)$ by increasing value of i and compute in $O(1)$ time the profitability difference between two consecutive values. This takes altogether $O(R)$ time per period. The intervals of profitabilities $\mathcal{U}(k_1,k_2)$ are computed altogether in $O(R \cdot T^3)$ time, by deducing $\mathbb{G}[k_1,k_2,i]$ from $\mathbb{G}[k_1,k_2-1,i]$ in $O(1)$ time.
In the fourth part, we compute the cost $H^*(k_1,k_2)$. For each k_1 and k_2, we compute and sort $\mathcal{U}(k_1,k_2)$ from the already sorted set $\mathcal{U}(k_1,k_2-1)$ in $O(R \cdot T)$ time. By Lemma 5.1.3, we can get $u^*(k_1,k_2)$ in $O(\log[R \cdot T])$ time using a binary search. Pre-computing the sums of disposable cost in Equation (5.10) takes $O(T^2 \cdot R)$ time. We deduce $H^*(k_1,k_2)$ in $O(1)$ time. The time complexity for the fourth part is hence $O(T^3 \cdot R)$.
Finally, we compute the optimal policy $f^*(T)$ in $O(T^2)$ time using dynamic programming. The total time complexity is thus $O(R \cdot T^3)$.

□

5.2 Solution without Disposables

In this section, we solve $DCPP$ when disposables are not allowed by reducing it to the dynamic lot-sizing problem. In Chapter 4, we described an algorithm which is optimal under the assumption that the demand is non-decreasing. When disposables are not allowed, the order quantity β_t at each period t is known:

$$\forall t: \ \beta_t := \sum_{r=0}^{R-1} D(t,r) \tag{5.11}$$

In addition, the number of outgoing full containers Z_t at each period is equal the total late demand from the previous period:

$$\forall t: \ Z_t := \sum_{r=R-L_{del}+1}^{R-1} D(t-1,r) \tag{5.12}$$

Lemma 5.2.1 *If after potentially purchasing at period t, a policy has less than $\beta_t + Z_t$ containers in the system, then this policy buys disposables.*

We use a different container assignment from the one in Chapter 4. We recall that a container is called busy whenever it is not idle, which includes the containers held by the supplier.

New container assignment $\mathcal{CA}_{ndis}$: We sequence the containers in the system and assign them to demand units such that the supplier orders the empty containers with the lowest sequence index.

Using container assignment $\mathcal{CA}_{ndis}$, we only order a a container for the first time when every container already used once are busy at the same period. We denote by $n_u(t)$ the number of containers used at least once from period 0 to period t. The other containers in the process at the same period have been idle since their purchase time.

Proposition 5.2.2 *Under the container assignment $\mathcal{CA}_{ndis}$, we have:*

$$\forall t \in [0, T-1] : \; n_u(t) = \max_{t' \in [0,t]} \left[Z_t + \beta_t \right] \tag{5.13}$$

We divide the cost of any policy without disposable into two costs:

1. The supplier and manufacturer holding costs relative to the $n_u(t)$ containers during each period t.
2. The purchasing cost of every container and the manufacturer cost of the containers above the required $n_u(t)$ at period t.

Since our system has no option to discard containers, the first group of costs is the same for every policy not using disposables, by Proposition 5.2.2. The problem of minimizing the second group of costs can be reduced to the following instance of dynamic lot-sizing problem:

- There are T time periods.
- The demand at time t is $n_u(t) - n_u(t-1)$.
- The purchasing cost of x containers at period is $C_{setup}(t) + x \cdot C_{cont}(t)$.
- The holding cost at period t is $C_{idle}(t)$.

Algorithm 5: Algorithm $Ndis.D$ using zero disposables

for $t : 0 \to T-1$ **do**
    Compute the best order size β_t;
    Compute the number of required containers $n_u(t)$;
Compute the best purchasing plan;
return the purchasing plan and the order policy;

Proposition 5.2.3 *If Hypothesis 4.3 [NoDisposable] holds, then Algorithm Ndis.D computes an optimal policy in $O(R \cdot T)$ time.*

Proof:

For each t, the values β_t and $n_u(t)$ can be computed in $O(R)$ time. We can use any of the fast algorithms from Federgruen and Tzur [27], wagelmans et al. [127], Aggarwal and Park [1] to solve each dynamic lot-sizing problem in $O(T)$ time and deduce the purchasing plan.

□

5.3 Algorithm Forbidding Close Placements

In this section, we adapt Algorithm *Flow*.4.1 to general demand patterns.

Hypothesis 5.2 ***[Idleness]:*** *In every optimal solution to DCPP, if k is a placement, then at least one container is idle at period k.*

Like Hypothesis 4.2 [Placement], this is a practicable hypothesis because a large amount of containers is purchased at each placement in a ramp-up scenario with high setup cost.

Remark 5.3.1 *Following our analysis from Section 4.5, we can assume without loss of generality that $t = 0$ and $t = T$ are placements.*

Remark 5.3.2 *There is no close plcaments under Hypothesis 5.2 [Idleness].*

For each $k \in [0, T]$, we define a network $\hat{\mathcal{F}}(k)$ from $\mathcal{G}$ as following (see Figure 5.2):

1. Remove every $v_{sup}(t, r)$, $v_{man}(t, r)$, $a_{ord}(t)$, $a_{sup}(t, r)$, $a_{pur}(t)$, $a_{dis}(t, r)$ such that $t \geq k$.
2. Remove every $v_{man}(t, r)$ and $a_{man}(t, r)$ so that $t \geq k + 1$.
3. Add an arc $a^{5.1}_{end}(k)$ from $v_{man}(k + 1, 0)$ to v_{pur} with negative cost $-C_{cont}(k) - C_{idle}(k)$.

Likewise, we create network $\hat{\mathcal{H}}(k_1, k_2)$ for $0 \leq k_1 < k_2 \leq T$ from $\hat{\mathcal{F}}(k_2)$ with the following steps (see Figure 5.3):

1. Remove every $v_{sup}(t, r)$, $v_{man}(t, r)$, $a_{man}(t, r)$, $a_{sup}(t, r)$ for all $t < k_1$.
2. Remove every ordering arc $a_{ord}(t)$ for $t \leq k_1$.
3. Remove every purchasing arc $a_{pur}(t)$ for $t \in [0, k_2[$.
4. Add an arc $a^{ord}_{pur}(k_1)$ from v_{pur} to $v_{sup}(k_1, 0)$ with cost $C_{cont}(k_1)$.

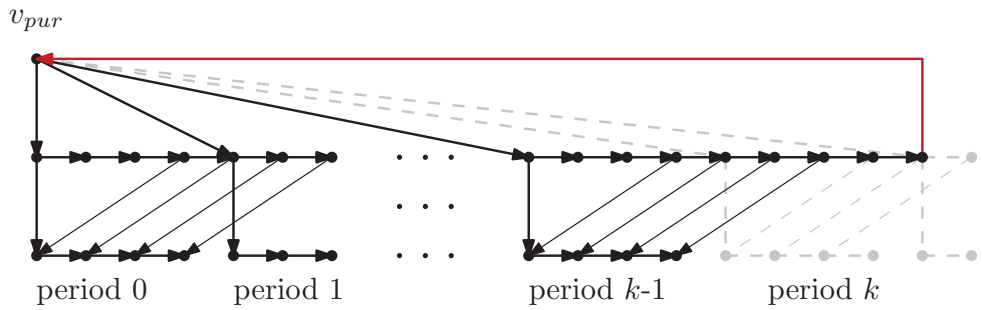


Figure 5.2: Network $\hat{\mathcal{F}}(k)$. Removed nodes and arcs are presented in gray. The added arc (in red) is $a^{5.1}_{end}(k)$.

5. Add an arc $a^{idle}_{pur}(k_1)$ from v_{pur} to $v_{man}(k_1+1, 0)$ with cost $C_{cont}(k_1) + C_{idle}(k_1)$.

6. For $r \in [0, L_{del}[$, set the excess of $v_{man}(k_1, r)$ to zero.

We note that step 6 sets the network $\hat{\mathcal{H}}(k_1, k_2)$ as if every late demand of period $k_1 - 1$ only uses disposables. We define $\hat{\mathcal{F}}(k_1, k_2)$ from $\hat{\mathcal{F}}(k_2)$ by removing every purchasing arc $a_{pur}(t)$ with $t \in]k_1, k_2[$. We use the same notations $\hat{f}^*$, $\hat{h}^*$, $\hat{F}^*$, $\hat{H}^*$ as in Chapter 4. By definition, we have:

$$\mathcal{G} = \hat{\mathcal{F}}(T) \tag{5.14}$$

Remark 5.3.3 *In contrast to Chapter 4, the networks $\hat{\mathcal{F}}(k_1, k_2)$, $\hat{\mathcal{F}}(k_2)$, and $\hat{\mathcal{H}}(k_1, k_2)$ do not include purchasing at period k_2, hence exclude the setup cost $C_{setup}(k_2)$.*

Finally, we define the network $\hat{\mathcal{G}}$ as the network containing every node and arc:

$$\hat{\mathcal{G}} = \mathcal{G} \cup \left\{ a^{ord}_{pur}(t), a^{idle}_{pur}(t), a^{5.1}_{end}(t) \ : \ t \in [0, T] \right\} \tag{5.15}$$

Lemma 5.3.4 *The networks $\hat{\mathcal{F}}(k)$, $\hat{\mathcal{H}}(k_1, k_2)$ and $\hat{\mathcal{F}}(k_1, k_2)$ are defined so that the excess in v_{pur} is zero.*

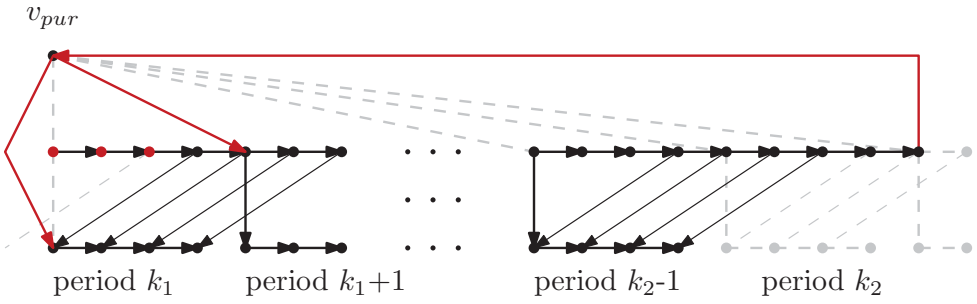


Figure 5.3: Network $\hat{\mathcal{H}}(k_1, k_2)$. Removed nodes and arcs are presented in gray. The added arcs (in red) are $a^{ord}_{pur}(k_1)$, $a^{idle}_{pur}(k_1)$ and $a^{5.1}_{end}(k_2)$. The nodes in red have a zero excess.

Proof:

For each supplier node, the manufacturer node corresponding to demand arrival is in the network with the opposite excess. In addition, each manufacturer node has a zero excess whenever the associated supplier node is not in the network.

□

Similarly to Chapter 4, we want to prove that we can compute the optimal solution $\hat{f}^*(T)$ to $DCPP$ by adding locally optimal flows $\hat{h}(k_1, k_2)$ and transposing the resulting flow into network $\mathcal{G}$.

Lemma 5.3.5 *For any flow f on any network in this section and for any period $r \in [0, T]$ and every time step $r \in [1, R[$ we have:*

$$f(a_{man}(t, r-1)) \leq f(a_{man}(t, r)) \tag{5.16}$$

Proof:

The flow in $a_{man}(t, r-1)$ arrives to $v_{man}(t, r)$ with excess $D(t, r-L_{del}) \geq 0$. The node $v_{man}(t, r)$ has one ingoing arc $a_{man}(t, r-1)$ and two outgoing arcs $a_{man}(t, r)$ and $a_{dis}(t, r-L_{del})$. Under Hypothesis 2.2 [Cost], at most $D(t, r-L_{del})$ flow units can go in arc $a_{dis}(t, r-L_{del})$. Therefore, the flow in $a_{man}(t, r)$ equals the flow in $a_{man}(t, r-1)$ plus the excess flow not going in $a_{dis}(t, r-L_{del})$.

□

We now create two operations, one to transform a flow on $\hat{\mathcal{F}}(k_1, k_2)$ into a flow on $\hat{\mathcal{F}}(k_1)$ plus a flow on $\hat{\mathcal{H}}(k_1, k_2)$, and the reverse operation. We call these operations the splitting and the merging operations respectively.

Given a flow on $\hat{\mathcal{G}}$, we define the *merging operation* at placement k as in Algorithm 6, removing the flow in the special arcs $a^{ord}_{pur}(k)$ and $a^{idle}_{pur}(k)$. In addition, it removes as much flow as possible from arc $a^{5.1}_{end}(k)$.

After using the merging operation on every placement of a flow, if there is no flow left on any arc $a^{5.1}_{end}(k)$, then the resulting flow is defined on network $\mathcal{G}$, hence is a solution to $DCPP$. We want to show that this condition holds under Hypothesis 5.2 [Idleness]. We call the merging operation *successful* if it removes the entire flow from the corresponding arc $a^{5.1}_{end}(k)$. We call a flow *feasible* if it only contains positive flow in arcs from network $\mathcal{G}$. Thus, a flow is feasible whenever it is generated by successful merging operations. We now analyze the construction of a flow merging locally optimal flows on networks $\hat{\mathcal{H}}(k_1, k_2)$.

Algorithm 6: Algorithm MergeFlow for placement k.

Data: Flow $\hat{f}$, placement k
// Step 1:
Add $\hat{f}\big(a^{ord}_{pur}(k)\big)$ flow units in $a_{ord}(k)$ and $a_{pur}(k)$;
Remove all flow in $\hat{f}\big(a^{ord}_{pur}(k)\big)$;
// Step 2:
Add $\hat{f}\big(a^{idle}_{pur}(k)\big)$ flow units in $a_{pur}(k)$ and $a_{man}(k,r)$ for all $r \in [0, R[$;
Remove all flow in $\hat{f}\big(a^{idle}_{pur}(k)\big)$;
// Step 3:
$x := \min\big\{\hat{f}\big(a_{man}(k,0)\big),\ \hat{f}\big(a_{pur}(k)\big),\ \hat{f}\big(a^{5.1}_{end}(k)\big)\big\}$;
Remove x flow units from arcs $a_{pur}(k)$, $a^{5.1}_{end}(k)$ and every arc $a_{man}(k,r)$ with $r \in [0, R[$;

Lemma 5.3.6 *The merging operation preserves the imbalance property on every node and the flow cost.*

Proof:

Algorithm 6 is decomposed into three steps. In the first step, we shift the flow from $a^{ord}_{pur}(k)$ to the path containing $a_{ord}(k)$ and $a_{pur}(k)$. In the second step, we shift the flow from $a^{idle}_{pur}(k)$ to the path containing $a_{pur}(k)$ and the arcs $a_{man}(k,r)$. In the third step, we remove some flow in the zero-cost cycle $v_{pur} \to v_{man}(k,0) \to \ldots \to v_{man}(k+1,0) \to v_{pur}$. By Lemma 5.3.5, every arc has a non-negative flow after the third step. Since we start from a flow and each step preserves the flow cost and the imbalance property, we conclude that the resulting pseudo-flow is a flow with the same cost.

□

We define the merging operation on two flows $\hat{f}(k_1)$ on $\hat{\mathcal{F}}(k_1)$ and $\hat{h}(k_1,k_2)$ at period k_1 as the merging operation of the sum of their flows transposed to $\hat{\mathcal{G}}$. We denote this operation as $\textbf{MergeFlow}\big(\hat{f}(k_1),\ \hat{h}(k_1,k_2)\big)$. Figure 5.4 shows the networks $\hat{\mathcal{F}}(k_1)$ and $\hat{\mathcal{H}}(k_1,k_2)$ around period k_1.

Proposition 5.3.7 *Consider locally optimal flows $\hat{f}^*(k_1)$ and $\hat{h}^*(k_1,k_2)$ on $\hat{\mathcal{F}}(k_1)$ and $\hat{\mathcal{H}}(k_1,k_2)$. The merging operation successfully transforms these flows into a flow $\hat{f}(k_1,k_2)$ on $\hat{\mathcal{F}}(k_1,k_2)$ with the same cost if and only if:*

$$\hat{f}(k_1)\big(a^{5.1}_{end}(k_1)\big) \le \hat{h}(k_1,k_2)\big(a^{idle}_{pur}(k_1)\big) + \min\big\{\hat{h}(k_1,k_2)\big(a^{ord}_{pur}(k_1)\big),\ \hat{f}(k_1)\big(a_{man}(k_1,0)\big)\big\} \quad (5.17)$$

Proof:

By Lemma 5.3.6, Algorithm 6 merges the initial flows into a single flow with the same total cost. The merging operation removes all the flow in arcs $a^{ord}_{pur}(k)$ and $a^{idle}_{pur}(k)$. Every other arc except $a^{5.1}_{end}(k)$ belongs to $\hat{\mathcal{F}}(k_1,k_2)$.

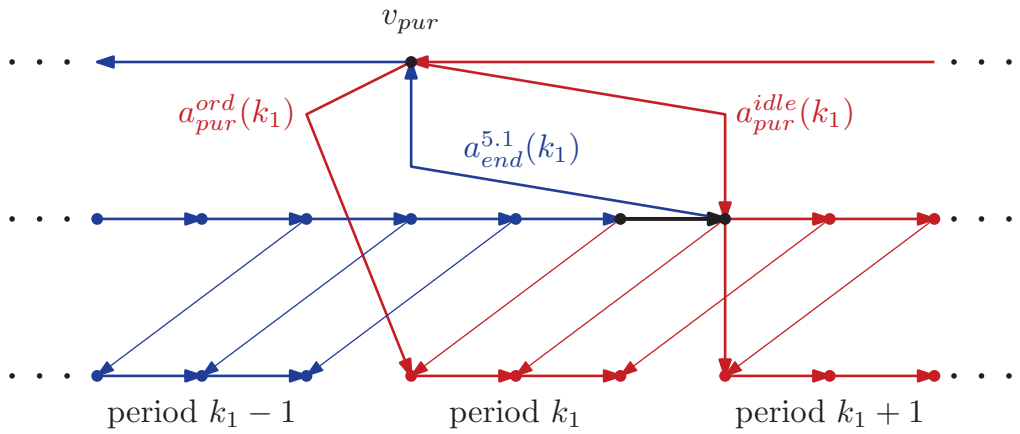


Figure 5.4: Netowkrs $\hat{\mathcal{F}}(k_1)$ and $\hat{\mathcal{H}}(k_1, k_2)$ around placement k_1. The blue nodes and arcs are in $\hat{\mathcal{F}}(k_1)$. The red nodes and arcs arcs are in $\hat{\mathcal{H}}(k_1, k_2)$. The black nodes and arc are common to both networks.

Consider the computation of x in the algorithm. The flow in $\hat{f}\big(a_{man}(k,0)\big)$ equals its initial value plus the initial flow in $a^{idle}_{pur}(k)$, and the flow in $\hat{f}\big(a_{pur}(k)\big)$ equals the initial flow in $a^{ord}_{pur}(k)$ plus the flow in $a^{idle}_{pur}(k)$, as its initial value is zero. Equation 5.17 holds if and only if both these terms are not lower than $\hat{f}(k_1)\big(a^{5.1}_{end}(k_1)\big)$, i.e. if and only if the flow in arc $a^{5.1}_{end}(k_1)$ is empty at the end of the algorithm. We conclude that the merging is successful and the resulting flow in defined on $\hat{\mathcal{F}}(k_1, k_2)$ if and only if Equation 5.17 holds.

□

Algorithm 7 describes the *splitting operation* as an inverse of the merging operation. The splitting operation removes the flow from arcs $a_{pur}(k)$ and $a_{pur}(k)$. By splitting a flow at every placement, we transpose a flow into several networks $\hat{\mathcal{H}}$.

Lemma 5.3.8 *Using the variable names from the splitting operation, we have that* $z \geq 0$.

Proof:

Since the input flow is feasible, every flow is non-negative and corresponds to the movement of containers in the system. In the algorithm, we have:

$$\begin{aligned} z &:= \sum_{t=0}^{k-1} \big(\hat{f}\big(a_{pur}(t)\big)\big) - \hat{f}\big(a^{5.1}_{end}(k)\big) \\ &= \sum_{t=0}^{k-1} \big(\hat{f}\big(a_{pur}(t)\big)\big) - x \\ &\geq - \big[\hat{f}\big(a_{ord}(k)\big) - \hat{f}\big(a_{pur}(k)\big)\big]^{+} \end{aligned}$$

Therefore, if $\hat{f}\big(a_{ord}(k)\big) \leq \hat{f}\big(a_{pur}(k)\big)$, then $z \geq 0$. Otherwise, note that the container fleet size $u^{\alpha}(k)$ after placement k is:

$$u^{\alpha}(k) := \sum_{t=0}^{k} \hat{f}\big(a_{pur}(t)\big)$$

Since we can only order containers which are present in the system, we have:

$$\begin{aligned}
&\sum_{t=0}^{k} \hat{f}\big(a_{pur}(t)\big) \geq \hat{f}\big(a_{ord}(k)\big) \\
&\Leftrightarrow \hat{f}\big(a_{pur}(k)\big) + z + \hat{f}\big(a^{5.1}_{end}(k)\big) \geq a_{ord}(k) \\
&\Leftrightarrow z \geq \hat{f}\big(a_{ord}(k)\big) - \hat{f}\big(a_{pur}(k)\big) - x \\
&\Leftrightarrow z \geq 0
\end{aligned}$$

We conclude that in both cases we get $z \geq 0$.

□

Algorithm 7: Algorithm SplitFlow for placement k.

Data: Feasible flow $\hat{f}$, placement k
// Step 1:
$x := \big[\hat{f}\big(a_{ord}(k)\big) - \hat{f}\big(a_{pur}(k)\big)\big]^{+}$;
Add x flow units to $a_{pur}(k)$, $a_{man}(k,r),\ r \in [0,R[$ and $a^{5.1}_{end}(k)$;
// Step 2:
Add $\hat{f}\big(a_{ord}(k)\big)$ flow units to $a^{ord}_{pur}(k)$;
Remove $\hat{f}\big(a_{ord}(k)\big)$ flow units from $a_{ord}(k)$ and $a_{pur}(k)$;
// Step 3:
$y := \min\{\hat{f}\big(a_{pur}(k)\big),\ \hat{f}\big(a_{man}(k,0)\big)\}$;
Add y flow units to $a^{idle}_{pur}(k)$;
Remove $\hat{f}\big(a_{pur}(k)\big)$ flow units from $a_{pur}(k)$ and $a_{man}(k,r),\ r \in [0,R[$;
// Step 4:
$z := \sum_{t=0}^{k-1} \big(\hat{f}\big(a_{pur}(t)\big)\big) - \hat{f}\big(a^{5.1}_{end}(k)\big)$;
Add z flow units to both $a^{5.1}_{end}(k)$ and $a^{idle}_{pur}(k)$;

Lemma 5.3.9 *The splitting operation preserves the imbalance property on every node and the flow cost.*

Proof:

Algorithm 7 is decomposed into four steps. In the first step, we add some flow to the zero-cost cycle containing $a_{pur}(k)$, $a_{man}(k,r)$ and $a^{5.1}_{end}(k)$ to ensure that the flow in $a_{pur}(k)$ is at least $\hat{f}\big(a_{ord}(k)\big)$. In the second step, we shift the flow from $a_{pur}(k)$ and $a_{ord}(k)$ to the equivalent arc $a^{ord}_{pur}(k)$. In the third step, we shift some flow y from $a^{idle}_{pur}(k)$ to the path containing $a_{pur}(k)$ and the $a_{man}(k,r)$. By Lemma 5.3.5, every arc has a non-negative flow after the third step. In the fourth step, we sdd some flow z to the zero-cost cycle composed of $a^{idle}_{pur}(k)$ and $a^{5.1}_{end}(k)$. By Lemma 5.3.8, the added flow is positive. Since we start from a flow and each step readily preserves the flow cost and the imbalance property, we conclude that the resulting pseudo-flow is a flow with the same cost.

□

Lemma 5.3.10 *Suppose that Hypothesis 2.2 [Cost] holds. Consider a flow f on $\hat{\mathcal{G}}$ with placement at period k. Let x denote the flow in arc $a_{man}(k, L_{ord}-1)$. After the splitting operation, for each $r \in [L_{ord}, R[$, the flow in arc $a_{man}(k,r)$ is at most x.*

Proof:

Consider $r \in [L_{ord}, R[$. The node $v_{man}(t,r)$ has excess $D(k, r-L_{del})$, one ingoing arc $a_{man}(k, r-1)$, and two outgoing arcs $a_{man}(k,r)$, $a_{dis}(k, r-L_{del})$. By Proposition 3.4.2, the flow in $a_{dis}(k, r-L_{del})$ is at most $D(k, r-L_{del})$. Therefore, the flow in $a_{man}(k,r)$ greater or equal to the flow in $a_{man}(k, r-1)$ and this lemma follows.

□

Proposition 5.3.11 *Suppose that Hypothesis 2.2 [Cost] holds. Consider a flow $\hat{f}(k_1,k_2)$ on $\hat{\mathcal{F}}(k_1,k_2)$ so that at period k_1 at least one container is idle and at least one is purchased. Using the splitting operation, we can decompose $\hat{f}(k_1,k_2)$ into a flow $\hat{f}(k_1)$ on $\hat{\mathcal{F}}(k_1)$ and a flow $\hat{h}(k_1,k_2)$ on $\hat{\mathcal{H}}(k_1,k_2)$ with the same total cost and such that:*

$$\hat{f}(k_1)\big(a^{5.1}_{end}(k_1)\big) \leq \hat{h}(k_1,k_2)\big(a^{idle}_{pur}(k_1)\big) \tag{5.18}$$

Proof:

In this proof, we decompose this flow into two pseudo-flows $\hat{f}(k_1)$ and $\hat{h}(k_1,k_2)$.

As shown in Figure 5.4, the only arcs which are common to the two networks $\hat{\mathcal{F}}(k_1)$ and $\hat{\mathcal{H}}(k_1,k_2)$ are $a_{man}(k_1,r)$ for $r \in [L_{ord}, R[$. By Lemma 5.3.10 in these arcs is at least equal to $a_{man}(k_1, L_{ord}-1)$. We decompose the flow in these arcs to that their flow in $\hat{f}(k_1)$ equals the flow in $a_{man}(k_1, L_{ord}-1)$. The left-over flows goes in $\hat{h}(k_1,k_2)$. Moreover, the two arcs $a_{pur}(k_1)$ and $a_{ord}(k_1)$ have zero flow after the splitting operation. Every other arc in

$\hat{\mathcal{F}}(k_1,k_2)$ is either in $\hat{\mathcal{F}}(k_1)$ or in $\hat{\mathcal{H}}(k_1,k_2)$. Thus, we add the flow in each of these arcs to the corresponding flow $\hat{f}(k_1)$ or $\hat{h}(k_1,k_2)$.
In network $\hat{\mathcal{F}}(k_1)$, the arcs $a_{man}(k_1,r)$ for $r \in [L_{ord}-1,R[$ have the same flow and can be contracted into a single arc a from node $v_{man}(k_1,L_{ord}-1)$ to v_{pur}. Besides this arc a and node v_{pur}, every node and every arc is either in $\hat{\mathcal{F}}(k_1)$ or in $\hat{\mathcal{H}}(k_1,k_2)$. By Lemma 5.3.9, the imbalance property is respected in the flow after the splitting operation. Consequently, every node in $\hat{\mathcal{F}}(k_1)$ respects the imbalance property besides maybe v_{pur}. By Lemma 5.3.4, the imbalance in $\hat{\mathcal{F}}(k_1)-\{v_{pur}\}$ is zero. Since v_{pur} is the only node connected to $\hat{\mathcal{F}}(k_1)-\{v_{pur}\}$ after the contraction of the flow in the arcs $a_{man}(k_1,r)$ for $r \in [L_{ord}-1,R[$ into arc a, it follows that v_{pur} respects the imbalance property in flow $\hat{f}(k_1)$. We deduce that both $\hat{f}(k_1)$ and $\hat{h}(k_1,k_2)$ are flows.

Since the flow $\hat{f}(k_1,k_2)$ does not contain the arcs $a^{idle}_{pur}(k_1)$ and $a^{5.1}_{end}(k_1)$, these arcs only contain the flow added by the splitting operation:

$$\hat{f}(k_1)\big(a^{5.1}_{end}(k_1)\big) = x+z$$
$$\hat{h}(k_1,k_2)\big(a^{idle}_{pur}(k_1)\big) = y+z$$

Using the algorithm variables, x flow units are added to both $a_{pur}(k_1)$ and $a_{man}(k_1,0)$ in step 1, then the flow in either $a_{pur}(k_1)$ or $a_{man}(k_1,0)$ is added to $a^{idle}_{pur}(k_1)$ in step 3, so we have $y \geq x$ and hence:

$$\hat{f}(k_1)\big(a^{5.1}_{end}(k_1)\big) \leq \hat{h}(k_1,k_2)\big(a^{idle}_{pur}(k_1)\big)$$

□

We define the splitting operation on flow $\hat{f}(k_1,k_2)$ on $\hat{\mathcal{F}}(k_1,k_2)$ as the splitting operation of this flow at period k_1. We denote by $\mathbf{SplitFlow}\big(\hat{f}(k_1,k_2)\big)$ this operation. That is, we assume that the splitting operation is by default on the last placement of the considered flow. The following result is a fundamental result in our study, and follows from Propositions 5.3.7 and 5.18.

Theorem 5.3.12 *Suppose that Hypothesis 2.2 [Cost] holds. Any feasible flow $\hat{f}(T)$ on $\mathcal{G}$ can be decomposed into n feasible flows $\hat{h}(k_i,k_{i+1})$, where $k_0=0<k_1<\ldots<k_{n-1}=T$ are the placements in $\hat{f}$. Moreover, these flows $\hat{h}(k_i,k_{i+1})$ are successfully merging together to generate $\hat{f}$ so that:*

$$\hat{F}(T) = \sum_{i=0}^{n-1}\big(\hat{H}(k_i,k_{i+1}) + C_{setup}(k_i)\big) \tag{5.19}$$

Theorem 5.3.12 justifies a first adaptation of Algorithm *Flow*.4.1 to general demand patterns:

Algorithm 8:

Initialize $\hat{f}^*(0)$ as an empty flow;
for $k_2 : 1 \to T$ **do**
  for $k_1 : 0 \to k_2 - 2$ **do**
    Compute $\hat{h}^*(k_1, k_2)$;
    $\hat{f}^*(k_1, k_2) :=$ MergeFlow$\big(\hat{f}^*(k_1),\ \hat{h}^*(k_1, k_2)\big)$;
  $\hat{f}^*(k_2) :=$ **feasible** flow $\hat{f}^*(k_1, k_2)$ with minimal $\hat{F}^*(k_1, k_2)$;
return $\hat{f}^*(T)$;

Theorem 5.3.13 *Suppose that Hypotheses 2.1 [Delay], 2.2 [Cost], and 4.4 [Distance] hold. Let $\mathcal{M}cf$ be the cost of a minimum linear-cost flow on any network $\hat{\mathcal{H}}(k_1, k_2)$. Algorithm 8 computes an optimal solution to DCPP in $O(T^2 \cdot \mathcal{M}cf)$ time if for every k_2 the flow $\hat{f}^*(k_1, k_2)$ minimizing $\hat{F}^*(k_1, k_2)$ is feasible.*

Proof:

The locally optimal flows $\hat{h}^*(k_1, k_2)$ can be computed altogether in $O(T^2 \cdot \mathcal{M}cf)$ time and the dynamic program takes $O(T^2)$ time, thus the complexity of Algorithm 8 is $O(T^2 \cdot \mathcal{M}cf)$.
By Hypothesis 4.4 [Distance], there is an optimal policy $\hat{f}^*(T)$ never purchasing at consecutive periods and let $k_0 = 0 < k_1 < \ldots < k_{n-1} = T$ be its placements. We denote by $\hat{h}^*(k_i, k_{i+1})$ and $\hat{f}^*(k_i, k_{i+1})$ the flow generated by splitting $\hat{f}^*(T)$ on its placements from k_{n-2} to k_1, and by $\hat{h}(k_i, k_{i+1})$ and $\hat{f}(k_i, k_{i+1})$ the flows generated by Algorithm 8.
We prove by recursion on i that $\hat{F}(k_i) \leq \hat{F}^*(k_i)$. This statement trivially holds for $i = 1$ as $\hat{f}(k_1) = \hat{h}(0, k_1)$ is locally optimal. Suppose now that the statement holds for all $j < i$. When Algorithm 8 computes $\hat{f}(k_1)$, it successfully merges the flows $\hat{f}(k_{i-1})$ and $\hat{h}(k_{i-1}, k_i)$, so the flow $\hat{f}(k_i)$ has at most cost $\hat{F}(k_{i-1} + \hat{H}(k_{i-1}, k_i))$. Since $\hat{h}(k_{i-1}, k_i)$ is locally optimal on $\hat{\mathcal{H}}(k_{i-1}, k_i)$ and by the recursion hypothesis, we deduce that $\hat{F}(k_i) \leq \mathcal{F}^*(k_i)$. Finally, Algorithm 8 computes a feasible policy with cost at most $\mathcal{F}^*(T)$. It is hence optimal and has cost $\mathcal{F}^*(T)$.

□

However, if Algorithm 8 encounters an unsuccessful merging flow $\hat{f}^*(k_1, k_2)$, this flow will not be selected for $\hat{f}^*(k_2)$ because it induces an infeasible policy. It is possible that the best policy up to period $k_2 - 1$ has period k_1 as last placement but using different flow.

Remark 5.3.14 *If we encounter an infeasible flow $\hat{f}^*(k_1, k_2)$, a simple way to improve the quality of Algorithm 8 would be to compute a minimum linear-cost flow on $\hat{\mathcal{F}}(k_1, k_2)$ instead of $\hat{\mathcal{H}}(k_1, k_2)$. We consider the same placements as in $\hat{f}^*(k_1)$ plus on placement at period k_1. This operation does not increase*

the complexity bound, as we compute at most $O(T^2)$ additional minimum linear-cost flows.

In the following, we develop an alternative algorithm which is optimal under Hypothesis 5.2 [Idleness].

Proposition 5.3.15 *Suppose that Hypotheses 2.1 [Delay], 2.2 [Cost], and 4.4 [Distance] hold. Let $\hat{f}^*(T)$ be an optimal solution to the problem respecting Hypothesis 5.2 [Idleness]. Using the splitting operation, $\hat{f}^*(T)$ can be decomposed into locally optimal flows $\hat{h}^*(k_1, k_2)$ on networks $\hat{\mathcal{H}}(k_1, k_2)$.*

Proof:

In this proof, we denote by $\hat{f}^*(k_1)$ the flow on $\hat{\mathcal{F}}(k_1)$ generated from the splitting operation on the flow $\hat{f}^*(k_2)$ at period k_1, where k_1 and k_2 are consecutive placements. Since we already know the placement sequence, we indifferently use the notations $\hat{f}^*(k_2)$ and $\hat{f}^*(k_1, k_2)$.
We prove this proposition by recursion. Consider the flow $\hat{f}^*(k_2)$ on $\hat{\mathcal{F}}(k_2)$ and suppose that it is locally optimal. At period k_1, at least one container is idle by assumption and at least one container is purchased by definition. By Proposition 5.3.11, the splitting operation decomposes into two flows $\hat{f}^*(k_1)$ and $\hat{h}^*(k_1, k_2)$ on $\hat{\mathcal{F}}(k_1)$ and $\hat{\mathcal{H}}(k_1, k_2)$.
Flow $\hat{f}^*(k_2)$ is locally optimal, so by Theorem 3.2.3 it does not contains any negative cycle. Any negative cycle in $\hat{h}^*(k_1, k_2)$ necessarily contains arc $a^{ord}_{pur}(k)$ or $a^{idle}_{pur}(k)$, as these are the only two arcs not in $\hat{\mathcal{F}}(k_2)$. Moreover, every other arc in the cycle has the same flow as in $\hat{f}^*(k_2)$. However, by construction, the flow in $a^{ord}_{pur}(k_1)$ is at most as big as the maximum of the flows in $a_{pur}(k_1)$ and $a_{ord}(k_1)$ on $\hat{\mathcal{F}}(k_2)$, whereas the flow in $a^{idle}_{pur}(k_1)$ is at most as big as the maximum of the flows in $a_{pur}(k_1)$ and $a_{man}(k_1, 0)$ on $\hat{\mathcal{F}}(k_2)$. Moreover, the cost of arc $a^{ord}_{pur}(k_1)$ (respectively $a^{idle}_{pur}(k_1)$) is equal to the cost of arcs $a_{pur}(k_1)$ and $a_{ord}(k_1)$ (respectively $a_{pur}(k_1)$ and $a_{man}(k_1, r), r \in [0, R[$). Thus, if there is a negative cycle in $\hat{h}^*(k_1, k_2)$, there is also a negative cycle in $\hat{f}^*(k_2)$. We conclude that flow $\hat{h}^*(k_1, k_2)$ is locally optimal.
Likewise, any negative cycle in $\hat{f}^*(k_1)$ must contain arc $a^{5.1}_{end}(k_1)$, and every other arc in the cycle has the same flow as in $\hat{f}^*(k_2)$. Therefore, for every negative cycle in $\hat{f}^*(k_1)$, there is a cycle in $\hat{f}^*(k_2)$ using arcs $a_{pur}(k_1)$ and $a_{man}(k_1, r), r \in [0, R[$, whose flows are positive by assumption and have the same cost. We deduce that $\hat{f}^*(k_1)$ does not contain any negative cycle and is hence locally optimal. The proposition follows by recursion on k_2.

□

Proposition 5.3.15 states that, under Hypothesis 5.2 [Idleness], an optimal flow $\hat{f}^*(T)$ can be decomposed into several locally optimal flows $\hat{h}^*(k_1, k_2)$. Nevertheless, this does not mean that a simple dynamic program as in Algorithm 8 automatically finds it. This issue comes from the merging operation not necessarily being successful.

Indeed, suppose that $k_1 < k_2 < k_3$ are three consecutive placements in the optimal policy. In order to compute the best flow $\hat{f}(k_2)$, a simple dynamic program will choose the previous placement k inducing the lowest cost $\hat{F}(k, k_2)$. Nevertheless, it may end up choosing another value $k \neq k_1$ inducing a cost $\hat{F}(k, k_2) \leq \hat{F}(k_1, k_2)$ but purchasing more containers so that the flow $\hat{f}(k, k_2)$ does not merge successfully with network $\hat{h}(k_2, k_3)$, whereas the flow $\hat{f}(k_1, k_2)$ does. In this case, a simple dynamic program has no way of recovering the placement k_1.
In the following, we update the dynamic program of Algorithm 8 to find an optimal solution. We start with the simple case where there is a unique locally optimal flow on every $\hat{\mathcal{H}}(k_1, k_2)$:

Hypothesis 5.3 ***[Unique-h*]:*** *For all $k_1 \in [0, T-2]$ and $k_2 \in [k_1+2, T]$, there is a unique locally optimal flow $\hat{h}^*(k_1, k_2)$.*

Under this hypothesis, Proposition 5.3.15 states that one combination of these flows $\hat{h}^*(k_1, k_2)$ builds an optimal solution to $DCPP$. We have defined the network $\hat{F}(k_1, k_2)$ so that every flow on it is feasible. So, we are looking for a minimum cost feasible flow on the network.

Lemma 5.3.16 *Suppose that Hypotheses 2.1 [Delay], 2.2 [Cost], 5.2 [Idleness], and 5.3 [Unique-h*] hold. Let $k_1 < k_2 < k_3$ be three successive placements. Then the flow in networks $\hat{h}^*(k_1, k_2)$ and $\hat{h}^*(k_2, k_3)$ determines whether the flows $\hat{f}^*(k_1, k_2)$ and $\hat{h}^*(k_2, k_3)$ can be successfully merged or not. It does not depend on $\hat{f}^*(k_1)$.*

Proof:

The success of the merging operation at period k_2 only depends on the arcs $a_{man}(k_2, 0)$, $a^{5.1}_{end}(k_2)$ from flow $\hat{f}^*(k_1, k_2)$ and arcs $a^{ord}_{pur}(k_2)$, $a^{idle}_{pur}(k_2)$ from flow $\hat{h}^*(k_2, k_3)$. Moreover, the flow in $a_{man}(k_2, 0)$ and $a^{5.1}_{end}(k_2)$ is the same in $\hat{f}^*(k_1, k_2)$ and in $\hat{h}^*(k_1, k_2)$, which proves this lemma.

□

Lemma 5.3.17 *Let $\hat{f}(k_1)$ be an infeasible flow on network $\hat{\mathcal{F}}(k_1) \cup \{a^{5.1}_{end}(t) : 0 \leq t < k_1\}$ and let $\hat{h}(k_1, k_2)$ be a flow on $\hat{\mathcal{H}}(k_1, k_2)$. Then the merging of these two flows is not feasible either.*

Proof:

The infeasibility of $\hat{f}(k_1)$ means that there is an arc $a^{5.1}_{end}(t)$ with positive flow where $t < k_1$. Since the merging operation does not affect the flow in any of these arcs, the merging does not generate a feasibe flow either.

□

Proposition 5.3.18 *Consider two successive placements $k_2 < k_3$ and suppose that Hypotheses 2.1 [Delay], 2.2 [Cost], 5.2 [Idleness], and 5.3 [Unique-h*] hold. The minimum cost flow $\hat{f}^*(k_2, k_3)$ equals the merging of the two flows $\hat{f}^*(k_1, k_2)$ and $\hat{h}^*(k_2, k_3)$ which can be successfully merged and have minimum cost over all $k_1 \in [0, k_2 - 2]$.*

Proof:

We show by recursion on k_2 that this proposition holds for every $k_1 < k_2 - 1$ and $k_3 > k_2 + 1$. For $k_2 = 2$ and for each $k_3 > k_2 + 1$,the only possible value for k_1 is 0, and $\hat{f}^*(k_1, k_2) = \hat{h}^*(k_1, k_2)$ is feasible and has minimum cost by definition.

Suppose that the statement holds for every $k < k_2$. Consider any $k_3 > k_2+1$. We look for the best flow $\hat{f}(k_2)$ so that $\hat{f}(k_2)$ and $\hat{h}^*(k_2, k_3)$ successfully merge and have minimum cost. By Lemma 5.3.17, $\hat{f}(k_2)$ must be feasible.

Let k_1 be the last placement in flow $\hat{f}(k_2)$, i.e. $\hat{f}(k_2) = \hat{f}(k_1, k_2)$, and denote by $\hat{f}(k_1)$ and $\hat{h}(k_1, k_2)$ the splitting of this flow. By Proposition 5.3.15, we only need to look at flows $\hat{f}(k_1, k_2)$ so that $\hat{h}(k_1, k_2) = \hat{h}^*(k_1, k_2)$ is locally optimal, as otherwise it cannot be part of an optimal solution.

By Lemma 5.3.16, it only depends on the flow $\hat{h}^*(k_1, k_2)$ whether $\hat{f}(k_1, k_2)$ and $\hat{h}^*(k_2, k_3)$ can merge. We deduce that $\hat{f}^*(k_2, k_3)$ must be the merging of two locally optimal solutions $\hat{f}^*(k_1, k_2)$ and $\hat{h}^*(k_2, k_3)$ with $k_1 \in [0, k_2 - 1]$, and the value k_1 inducing a minimum cost among the values inducing a successful merging.

□

Algorithm *Flow*.5.1 presents the adapted dynamic program. We use the notation $\hat{f}^*(k_0, k_1, k_2)$ for the minimum cost feasible flow on network $\hat{F}(k_2)$ so that k_0 and k_1 are the last two placements, and denote by $\hat{F}^*(k_0, k_1, k_2)$ its cost.

Theorem 5.3.19 *We suppose that Hypotheses 2.1 [Delay], 2.2 [Cost], 5.2 [Idleness], and 5.3 [Unique-h*] hold. Then Algorithm Flow*.5.1 *solves the DCPP optimally in $O(T^2 \cdot \mathcal{M}cf)$ time, where $\mathcal{M}cf$ is the time complexity of a minimum linear-cost flow on any network $\hat{\mathcal{H}}(k_1, k_2)$.*

Proof:

The optimality follows from Proposition 5.3.18. The complexity of computing every flow $\hat{h}(k_1, k_2)$ is $O(T^2 \cdot \mathcal{M}cf)$, and the dynamic program takes $O(T^3)$ time. Since the networks $\hat{\mathcal{H}}(k_1, k_2)$ have $O(R \cdot T)$ sources and $O(R \cdot T)$ sinks, the minimum complexity of any algorithm is $O(R \cdot T)$ time. We conclude that the time complexity is bounded by $O(T^2 \cdot \mathcal{M}cf)$.

□

Algorithm 9: Algorithm *Flow*.5.1

Initialize $\hat{f}^*(0)$ as an empty flow;
for $k_2 : 1 \to T$ **do**
  Compute $\hat{h}^*(0, k_2)$;
  $\hat{f}^*(0, k_2) := \hat{h}^*(0, k_2)$;
  $\hat{f}^*(0, 0, k_2) := \hat{h}^*(0, k_2)$;
  for $k_1 : 0 \to k_2 - 2$ **do**
    Compute $\hat{h}^*(k_1, k_2)$;
    for $k_0 : 0 \to k_1 - 2$ **do**
      $\hat{f}^*(k_0, k_1, k_2) := \text{MergeFlow}\big(\hat{f}^*(k_0, k_1),\ \hat{h}^*(k_1, k_2)\big)$;
    $\hat{f}^*(k_1, k_2) :=$ **feasible** flow $\hat{f}^*(k_0, k_1, k_2)$ minimizing $\hat{F}^*(k_0, k_1, k_2)$;
$\hat{f}^*(T) :=$ flow $\hat{f}^*(k_1, T)$ minimizing $\hat{F}^*(k_1, T)$;
return $\hat{f}^*(T)$;

Remark 5.3.20 *Proposition 5.3.18 still holds if we assume, instead of Hypothesis 5.2 [Idleness], that at least one optimal policy has at least one idle container after every placement.*

Since Algorithm *Flow*.5.1 does the same as Algorithm 8 but considers the best policy over $O(T)$ times more candidates, the new dynamic program always generates a better policy and the optimality certificate is still valid for this algorithm.

Theorem 5.3.21 *We suppose that Hypotheses 2.1 [Delay], 2.2 [Cost], and 4.4 [Distance] hold. The cost of the policy generated by Algorithm Flow*.5.1 *is never greater than the cost of the policy generated by Algorithm 8. Moreover, Algorithm Flow*.5.1 *computes an optimal policy if the flow* $\hat{f}^*(k_1, k_2)$ *minimizes the cost over every* $\hat{f}^*(k_0, k_1, k_2)$.

We finally consider several locally optimal flows on $\hat{\mathcal{H}}(k_1, k_2)$, and show that Algorithm *Flow*.5.1 still computes an optimal policy.

Lemma 5.3.22 *We suppose that Hypotheses 2.1 [Delay], 2.2 [Cost], and 5.2 [Idleness] hold. Consider a flow* $\hat{f}(T)$ *on* $\mathcal{G}$ *and suppose it has been generated using some feasible flows* $\hat{h}(k_0, k_1)$ *and* $\hat{h}(k_1, k_2)$, *which is possible by Theorem 5.3.12. Let* X *be the following value, which is non-negative by Proposition 5.3.7:*

$$\begin{aligned} X :=& \hat{h}(k_1, k_2)\big(a^{idle}_{pur}(k_1)\big) - \hat{h}(k_0, k_1)\big(a^{5.1}_{end}(k_1)\big) \\ &+ \min\big\{\hat{h}(k_1, k_2)\big(a^{ord}_{pur}(k_1)\big),\ \hat{h}(k_0, k_1)\big(a_{man}(k_1, 0)\big)\big\} \geq 0 \end{aligned} \tag{5.20}$$

Then X corresponds to the minimum between the number of purchased containers and the number of idle containers at period k_1. If $\hat{f}(T)$ is an optimal flow then X is strictly positive.

Proof:

By construction, the number of idle (respectively purchased) containers at period k_1 is equal to the flow in arc $a_{man}(k_1, 0)$ (respectively $a_{pur}(k_1)$) in the flow generated by merging $\hat{h}(k_0, k_1)$ and $\hat{h}(k_1, k_2)$.
By Theorem 5.3.12, the feasible flows $\hat{h}(k_0, k_1)$ and $\hat{h}(k_1, k_2)$ successfully merge. Consider Algorithm 6 after step 2. The flow contains a zero cost cycle whose flow is $x := \min\left\{\hat{f}\big(a_{man}(k, 0)\big),\ \hat{f}\big(a_{pur}(k)\big),\ \hat{f}\big(a^{5.1}_{end}(k)\big)\right\}$. We have shown in Proposition 5.3.7 that if $X \geq 0$, then the minimum value is $\hat{f}\big(a^{5.1}_{end}(k)\big)$. Step 3 of the merging algorithm removes this zero-cost cycle and the merged flow in either $a_{pur}(k_1)$ or $a_{man}(k_1, 0)$ is X. However, by assumption, k_1 is a placement in $\hat{f}(T)$, so the flow in $a_{pur}(k_1)$ is positive. Moreover, since $\hat{f}(T)$ is optimal, we have by Hypothesis 5.2 [Idleness] that the flow in $a_{man}(k_1, 0)$ is also positive. We conclude that $X > 0$

□

Proposition 5.3.23 *Consider two optimal flows f_1 and f_2 on a network. We can generate f_2 from f_1 by adding exclusively zero-cost cycles.*

Proof:

Firstly, we can prove by induction on the number of changed arcs that we can transform any flow to any other flow by adding cycles. Indeed, if we add a path from a node v_1 to another node v_2, then we create a pseudo-flow where the balance is not respected for nodes v_1 and v_2 until we add a path from node v_2 to node v_1 with the same flow quantity.
Suppose now that to generate f_2 from f_1 we need to add non-zero cost cycles. In the following, we exclusively consider cycles used for this transformation. We define the *absorbing set of a cycle* by firstly putting this cycle (alone) in the set, then iteratively adding to the set every cycle sharing at least one arc with at least one cycle already in the set.
We consider the absorbing set of a negative cycle. Since this set is independent from any other cycle, we can add this cycle to f_2. We prove by induction on the number of arcs that we can transform it into a set of zero-cost cycles. Since there is a negative-cost cycle, at least one arc must be considered more than once. Otherwise, there would be no negative cycle in it. Consider an arc visited at least twice. Necessarily the flow in both direction must be the same. We can hence divide the cycle into one zero-cost cycle containing this arc in the two directions, and every other arc. These leftover arcs contain one less arc, so if they still contain a negative cycle, we deduce from the induction hypothesis that we can transform them into a set of zero-cost cycles.

□

Corollary 5.3.24 *We suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Consider three periods k_0, k_1, k_2 so that $k_0 \in [0, T-4]$, $k_1 \in [k_0+2, T-2]$ and $k_2 \in [k_1+2, T]$. Suppose that there are two locally optimal flows $\hat{h}_1(k_0,k_1)$ and $\hat{h}_2(k_0,k_1)$ on $\hat{\mathcal{H}}(k_0,k_1)$ and two locally optimal flows $\hat{h}_1(k_1,k_2)$ and $\hat{h}_2(k_1,k_2)$ on $\hat{\mathcal{H}}(k_1,k_2)$ so that $\hat{h}_1(k_0,k_1)$ and $\hat{h}_1(k_1,k_2)$ successfully merge, whereas $\hat{h}_2(k_0,k_1)$ and $\hat{h}_2(k_1,k_2)$ do not. Then there exist two locally optimal flows $\hat{h}_3(k_0,k_1)$ and $\hat{h}_3(k_1,k_2)$ which successfully merge and so that the merged flow has zero flow in arc $a_{man}(k_1,0)$.*

Proof:

Let $\hat{h}_1(k_0,k_1,k_2)$ and $\hat{h}_2(k_0,k_1,k_2)$ be the two merged flows on $\hat{\mathcal{G}}$. By Proposition 5.3.23, we can transform $\hat{h}_1(k_0,k_1,k_2)$ into $\hat{h}_2(k_0,k_1,k_2)$ by adding one-flow-unit zero-cost cycles on $\hat{\mathcal{G}}$. These cycles change the flow on each arc by at most one, and therefore change the value X described in Lemma 5.3.22 by at most one. Since k_1 is a placement, we can assume without loss of generality that $a_{pur}(k_1)$ must be positive. Since $\hat{h}_1(k_0,k_1,k_2)$ is so that $X \geq 0$ and $\hat{h}_2(k_0,k_1,k_2)$ is so that $X < 0$, there exists necessarily a flow $\hat{h}_3(k_0,k_1,k_2)$ so that $X = 0$ which can be split into two locally optimal flows $\hat{h}_3(k_0,k_1)$ and $\hat{h}_3(k_1,k_2)$. These locally optimal flows merge successfully and induce a zero flow in $a_{man}(k_1,0)$ by Lemma 5.3.22.

□

Theorem 5.3.25 *We suppose that Hypotheses 2.1 [Delay], 2.2 [Cost], and 5.2 [Idleness] hold. Then Algorithm Flow.5.1 solves DCPP optimally in $O(T^2 \cdot \mathcal{M}cf)$ time, where $\mathcal{M}cf$ is the time complexity of a minimum linear-cost flow on any network $\hat{\mathcal{H}}(k_1,k_2)$.*

Proof:

The complexity is the same as for Theorem 5.3.19. By Proposition 5.3.15, any optimal flow $\hat{f}^*(T)$ can be decomposed into locally optimal flows $\hat{h}^*(k_i,k_{i+1})$ with $0 = k_0 < k_1 < \ldots < k_{n-1} = T$.
Algorithm *Flow*.5.1 successively generates the flows $\hat{f}(k_i,k_{i+1})$ using the locally optimal flows $\hat{h}(k_i,k_{i+1})$. If every best flow is feasible, then by Theorem 5.3.21 the generated policy is optimal. Using the same arguments, if for every $i \in [2,n[$ the flow $\hat{f}(k_{i-2},k_{i-1},k_i)$ is feasible, then the final policy will have at most cost $\hat{F}(T)$, and hence will be optimal.
Otherwise, consider the first placement $k_i \in \{k_0,\ldots,k_{n-1}\}$ so that the flow $\hat{f}(k_{i-2},k_{i-1},k_i)$ is not feasible. Consequently, $\hat{f}(k_{i-2},k_{i-1})$ is feasible but cannot successfully merge with $\hat{h}(k_{i-1},k_i)$. Using Proposition 5.3.7, we deduce that the flows $\hat{h}(k_{i-2},k_{i-1})$ and $\hat{h}(k_{i-1},k_i)$ do not merge successfully. However, the flows $\hat{h}^*(k_{i-2},k_{i-1})$ and $\hat{h}^*(k_{i-1},k_i)$ defined on the same networks $\hat{\mathcal{H}}(k_{i-2},k_{i-1})$ and $\hat{\mathcal{H}}(k_{i-1},k_i)$ successfully merge. By Corollary 5.3.24,

there are two locally optimal flows $\hat{h}_3(k_{i-2}, k_{i-1})$ and $\hat{h}_3(k_{i-1}, k_i)$ successfully merging and so that the flow in $a_{man}(k_{i-1}, 0)$ is zero.
The two $\hat{f}(k_{i-2}, k_{i-1})$ and $\hat{h}_3(k_{i-1}, k_i)$ necessarily successfully merge due to $\hat{f}(k_{i-2}, k_{i-1})$ and $\hat{h}_3(k_{i-1}, k_i)$ successfully merging. Indeed, $\hat{h}_3(k_{i-1}, k_i)$ has been generated from $\hat{h}(k_{i-1}, k_i)$ by purchasing more containers and $\hat{h}_3(k_i, k_{i-1})$ from $\hat{h}_3(k_i, k_{i-1})$ by purchasing less containers.
We prove by recursion on k_i that there exists a flow $\hat{f}_3(k_{i-1}, k_i)$ with cost $\hat{F}(k_{i-1}, k_i) \leq \hat{F}^*(k_{i-1}, k_i)$ and so that no container is idle after at least one placement. For each coming placement $k_j > k_i + 1$, either $\hat{f}_3(k_{j-2}, k_{j-1})$ successfully merges with $\hat{h}(k_{j-1}, k_j)$ hence generating $\hat{f}_3(k_{j-1}, k_j)$, or we use Corollary 5.3.24 to show that there are two other locally optimal flows $\hat{h}_4(k_{j-2}, k_{j-1})$ and $\hat{h}_4(k_{j-1}, k_j)$ which successfully merge together, successfully merge with $\hat{f}_3(k_{j-3}, k_{j-2})$ and so that no container is idle at period k_{j-1}.
When we arrive at $k_j = k_{n-1} = T$, we have proven that there is an optimal policy violating Hypothesis 5.2.

□

Remark 5.3.26 *The enhanced capacity scaling algorithm presented in the previous chapters requires the network to have exclusively non-negative arcs. However, arc $a^{5.1}_{end}(k)$ has a strictly negative cost $-C_{cont}(k) - C_{idle}(k)$. Ahuja et al. [2] propose the following network update to have only non-negative costs. Firstly, we compute an upper-bound of the flow which may go in each negative-cost arc, for example the total positive excess of every source in the network. We then send this amount of flow in the arc and consider the residual network. In the residual network, this arc is inverted and has positive cost. Note that we must not forget to take into account to cost of this initial amont of flow.*

5.4 Algorithm Allowing Close Placements

Algorithm *Flow*.5.1 does not allow placements at consecutive periods. We now extend it to allow close placements. We consider the following hypothesis:

Hypothesis 5.4 ***[Idleness-Or-Placement]:*** *There is an optimal solution to DCPP so that if period k is a placement, then either:*

- *a container is idle during period k, or*
- *period $k+1$ is a placement.*

We do not prove all results rigorously as they can be deduced with similar arguments as in the previous section, for Algorithm *Flow*.5.1.

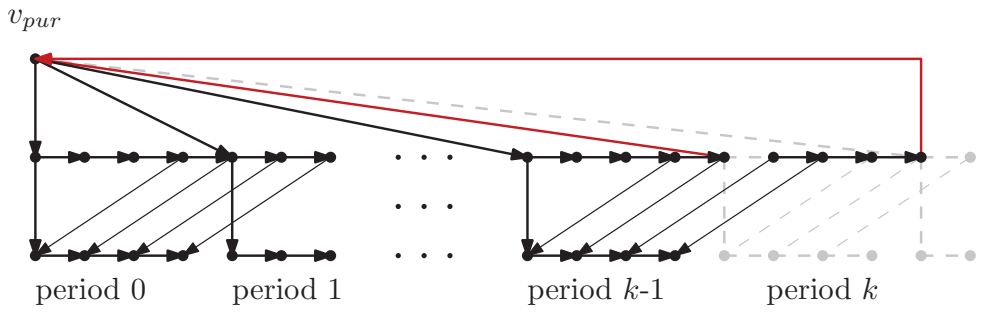


Figure 5.5: Network $\hat{\mathcal{F}}_2(k)$. Removed nodes and arcs are presented in gray, and added arcs are in red.

We create networks $\hat{\mathcal{F}}_2(k_2)$, $\hat{\mathcal{H}}_2(k_1,k_2)$, and $\hat{\mathcal{F}}_2(k_1,k_2)$ from respectively $\mathcal{F}(k_2)$, $\mathcal{H}(k_1,k_2)$, and $\mathcal{F}(k_1,k_2)$ by (see Figures 5.5 and 5.6):

1. replacing arcs $a_{man}(k_2,0), r \in [0,R[$ with a new arc $a^{5.2}_{end}(k_2)$ from node $v_{man}(k_2,0)$ to v_{pur} with cost $-C_{cont}(k_2)$, and
2. replacing arc $a^{5.1}_{end}(k_2)$ with a new arc $a^{5.2}_{end}(k_2+1)$, which is also from node $v_{man}(k_2+1,0)$ to node v_{pur} but with cost $-C_{cont}(k_2+1)$.

Note that for every t, arcs $a^{5.2}_{end}(k_2)$ and $a_{pur}(k_2)$ build a zero-cost cycle. Since we now consider close placements, we define the network $\hat{\mathcal{H}}(k_1,k_2)$ for $k_1 = k_2 - 1$. Network $\hat{\mathcal{H}}(k_2-1,k_2)$ has a similar definition as other networks $\hat{\mathcal{H}}(k_1,k_2)$. The only difference is that we do not need the arc $a^{idle}_{pur}(k_2-1)$, because no container will be idle at period k_2-1 by Hypothesis 5.4 [Idleness-Or-Placement].
Figure 5.7 illustrates the networks $\hat{\mathcal{F}}_2(k_2-1)$ and $\hat{\mathcal{H}}(k_2-1,k_2)$ around period k_2-1 for consecutive placements $k_1 := k_2-1$ and k_2. For each $k_2 \in [2,T]$ and $k_1 \in [0,k_1-1]$, any flow on $\hat{\mathcal{F}}(k_1,k_2)$ can be split into:

- a flow on $\hat{\mathcal{F}}(k_1)$ and a flow on $\hat{\mathcal{H}}(k_1,k_2)$ if $k_1 < k_2 - 1$, or
- a flow on $\hat{\mathcal{F}}_2(k_1)$ and a flow on $\hat{\mathcal{H}}_2(k_1,k_2)$ if $k_1 = k_2 - 1$.

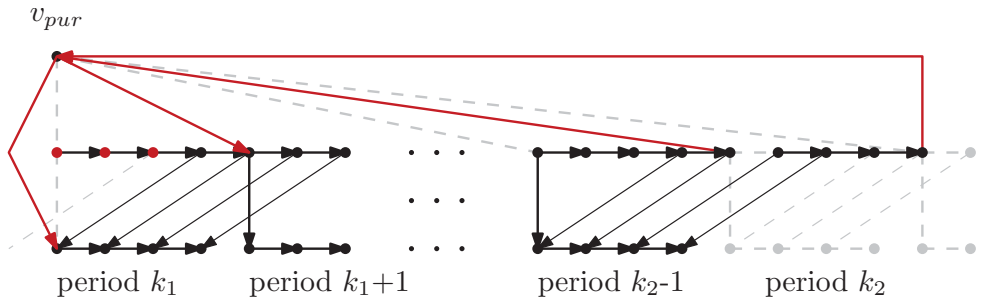


Figure 5.6: Network $\hat{\mathcal{H}}_2(k_1,k_2)$. Removed nodes and arcs are presented in gray, and added arcs are in red.

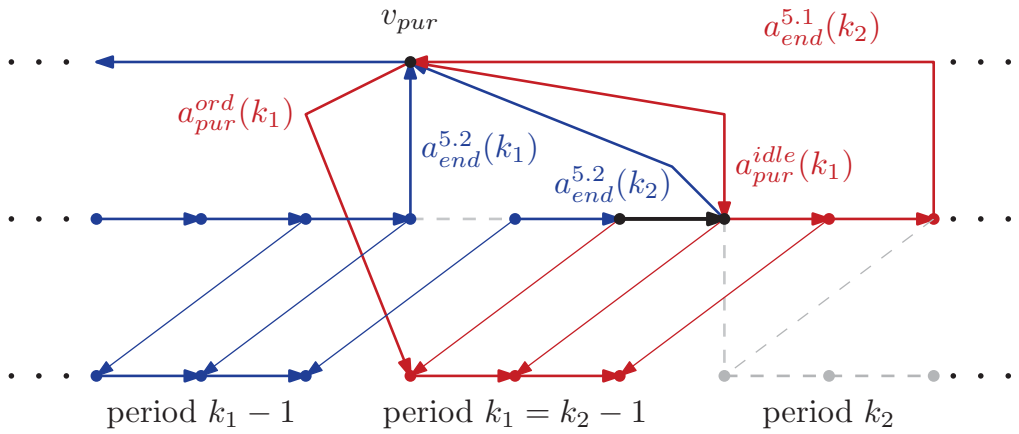


Figure 5.7: Flows $\hat{f}_2(k_1)$ and $\hat{h}(k_1, k_2)$ around placement k_1, when $k_1 = k_2 - 1$ and the placement after k_2 is $k_3 > k_2 + 1$. The blue nodes and arcs are in $\hat{\mathcal{F}}_2(k_1)$. The red nodes and arcs arcs are in $\hat{\mathcal{H}}(k_1, k_2)$. The black nodes and arc are common to both networks and gray arcs are in none.

Moreover, any flow on $\hat{\mathcal{F}}_2(k_1, k_2)$ can be split into:

- a flow on $\hat{\mathcal{F}}(k_1)$ and a flow on $\hat{\mathcal{H}}_2(k_1, k_2)$ if $k_1 < k_2 - 1$, or
- a flow on $\hat{\mathcal{F}}_2(k_1)$ and a flow on $\hat{\mathcal{H}}_2(k_1, k_2)$ if $k_1 = k_2 - 1$.

For the merging of two flows $\hat{f}(k)$ and $\hat{h}(k-1, k)$ to be successful, we require in addition arc $a^{ord}_{pur}(k-1)$ to have a bigger flow than $a^{5.2}_{end}(k-1)$. Nonetheless, if the flow in $a^{5.2}_{end}(k-1)$ was greater than the flow in $a^{ord}_{pur}(k-1)$, we could decrease the solution cost by not purchasing at period $k-1$. Therefore, it is not restrictive to assume than the flow in $a^{ord}_{pur}(k-1)$ is bigger. Similarly to Algorithm 8, we extend the ZIO property and deduce Algorithm 10.

Algorithm 10:

Initialize $\hat{f}^*(0)$ as an empty flow;
for $k_2 : 1 \to T$ **do**
  for $k_1 : 0 \to k_2 - 2$ **do**
    Compute $\hat{h}^*(k_1, k_2)$;
    $\hat{f}^*(k_1, k_2) :=$ MergeFlow$\big(\hat{f}^*(k_1),\ \hat{h}^*(k_1, k_2)\big)$;
    $\hat{f}_2{}^*(k_1, k_2) :=$ MergeFlow$\big(\hat{f}^*(k_1),\ \hat{h}_2{}^*(k_1, k_2)\big)$;
  Compute $\hat{h}^*(k_2 - 1, k_2)$;
  $\hat{f}^*(k_2 - 1, k_2) :=$ MergeFlow$\big(\hat{f}_2{}^*(k_2 - 1),\ \hat{h}^*(k_2 - 1, k_2)\big)$;
  $\hat{f}_2{}^*(k_2 - 1, k_2) :=$ MergeFlow$\big(\hat{f}_2{}^*(k_2 - 1),\ \hat{h}_2{}^*(k_2 - 1, k_2)\big)$;
  $\hat{f}^*(k_2) :=$ **feasible** flow $\hat{f}^*(k_1, k_2)$ with minimal $\hat{F}^*(k_1, k_2)$;
  $\hat{f}_2{}^*(k_2) :=$ **feasible** flow $\hat{f}_2{}^*(k_1, k_2)$ with minimal $\hat{F}_2{}^*(k_1, k_2)$;
return $\hat{f}^*(T)$;

Theorem 5.4.1 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. If the best feasible flows $\hat{f}(k_2)$ and $\hat{f}_2(k_2)$ always minimize the cost, i.e. if we never encounter a k_1 minimizing $\hat{F}^*(\cdot, k_2)$ or $\hat{F_2}^*(\cdot, k_2)$ while inducing an infeasible flow, then Algorithm 10 generates an optimal policy.*

Using the dynamic program from Algorithm *Flow*.5.1, we extend Algorithm 10 to Algorithm *Flow*.5.2 to compute an optimal policy under Hypothesis 5.4 [Idleness-Or-Placement].

Algorithm 11: Algorithm *Flow*.5.2

Initialize $\hat{f}^*(0)$ as an empty flow;
for $k_2 : 1 \to T$ **do**
  for $k_1 : 0 \to k_2 - 1$ **do**
    Compute $\hat{h}^*(k_1, k_2)$;
    if $k_1 == 0$ **then**
      $\hat{f}^*(0, k_2) := \hat{h}^*(0, k_2)$;
      $\hat{f}^*(0, 0, k_2) := \hat{h}^*(0, k_2)$;
      $\hat{f_2}^*(0, k_2) := \hat{h}^*(0, k_2)$;
      $\hat{f_2}^*(0, 0, k_2) := \hat{h}^*(0, k_2)$;
    else
      for $k_0 : 0 \to k_1 - 1$ **do**
        if $k_1 == k_2 - 1$ **then**
          $\hat{f}^*(k_0, k_1, k_2) := \text{MergeFlow}\big(\hat{f}^*(k_0, k_1),\ \hat{h}^*(k_1, k_2)\big)$;
          $\hat{f_2}^*(k_0, k_1, k_2) :=$
           $\text{MergeFlow}\big(\hat{f}^*(k_0, k_1),\ \hat{h_2}^*(k_1, k_2)\big)$;
        else
          $\hat{f}^*(k_0, k_1, k_2) := \text{MergeFlow}\big(\hat{f_2}^*(k_0, k_1),\ \hat{h}^*(k_1, k_2)\big)$;
          $\hat{f_2}^*(k_0, k_1, k_2) :=$
           $\text{MergeFlow}\big(\hat{f_2}^*(k_0, k_1),\ \hat{h_2}^*(k_1, k_2)\big)$;
      $\hat{f}^*(k_1 - 1, k_1, k_2) := \text{MergeFlow}\big(\hat{f}^*(k_1 - 1, k_1),\ \hat{h}^*(k_1, k_2)\big)$;
      $\hat{f}^*(k_1, k_2) :=$ **feasible** flow $\hat{f}^*(k_0, k_1, k_2)$ with minimal $\hat{F}^*(k_0, k_1, k_2)$;
      $\hat{f_2}^*(k_1, k_2) :=$ **feasible** flow $\hat{f_2}^*(k_0, k_1, k_2)$ with minimal $\hat{F_2}^*(k_0, k_1, k_2)$;
$\hat{f}^*(T) :=$ flow $\hat{f}^*(k_1, T)$ minimizing $\hat{F}^*(k_1, T)$;
$\hat{f_2}^*(T) :=$ flow $\hat{f_2}^*(k_1, T)$ minimizing $\hat{F}^*(k_1, T)$;
return Flow among $\hat{f}^*(T)$ or $\hat{f_2}^*(T)$ with minimal cost;

Theorem 5.4.2 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. The cost of the policy generated by Algorithm Flow.5.2 is never greater than the cost of the policy generated by Algorithm 10. Moreover, Algorithm Flow.5.2 computes an optimal policy if the best flows $\hat{f}^*(k_1, k_2)$ and $\hat{f}_2^*(k_1, k_2)$ minimize the cost over every $\hat{f}^*(k_0, k_1, k_2)$ and $\hat{f}_2^*(k_0, k_1, k_2)$ respectively.*

Theorem 5.4.3 *Suppose that Hypotheses 2.1 [Delay], 2.2 [Cost], and 5.4 [Idleness-Or-Placement] hold. Algorithm Flow.5.2 solves DCPP optimally in $O(T^2 \cdot \mathcal{M}cf)$ time, where $\mathcal{M}cf$ is the time complexity of a minimum linear-cost flow on any network $\hat{\mathcal{H}}(k_1, k_2)$.*

5.5 Comparison and Extensions

5.5.1 Comparison

In Chapter 4, we have constructed two algorithms generating an optimal solution under some hypotheses in the case of a non-decreasing demand. Chapter 5 proposes alternative algorithms which are optimal under similar hypotheses but for any demand pattern. These algorithms all have the same structure, namely first running $O(T^2)$ minimum linear-cost flow algorithms, then using a dynamic program to build a solution from these flows.

There is a fundamental difference between the algorithms from the two chapters. In Chapter 4, the solution $\hat{h}^*(k_1, k_2)$ has a fixed amount of containers at periods k_2 and $k_2 + 1$. The question is whether these containers should come at period k_2 or before. The solution is infeasible if we start at period $k_1 + 2$ with more containers than this amount, due to arc $a_{end}(k_2 + 1)$ not being empty.

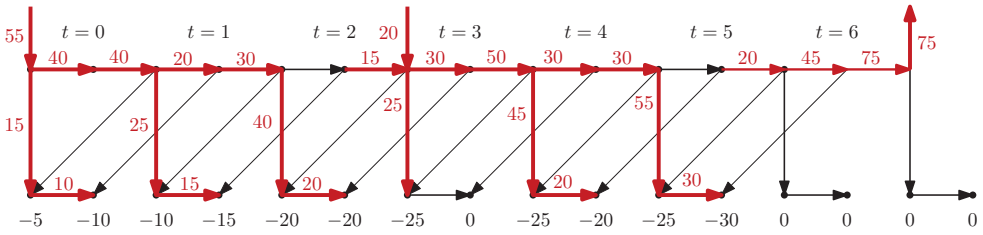
On the other hand, in this chapter we purchase at period k_1 as many containers as it is profitable to, by virtually selling every container at period $k_2 + 1$. We purchase these containers again in the next flow network $\hat{\mathcal{H}}(k_2, k_3)$. The solution may be infeasible if the number of containers we need in period interval $[k_2, k_3[$ is not significantly greater than the number of containers in period interval $[k_1, k_2[$.

Figure 5.8 shows an example of a network where Algorithms *Flow*.4.1 and *Flow*.4.2 compute a sub-optimal policy whereas Algorithms *Flow*.5.1 and *Flow*.5.2 compute an optimal policy. This example has zero setup cost at periods 0 and 3, and 1000 setup cost at other periods, so that only periods 0 and 3 are placements. Moreover, the disposable cost is very high such that every optimal policy necessarily fulfills every demand. Algorithm *Flow*.4.1 cannot purchase more than 45 containers at period $t = 0$, which represents the number of containers we need to fulfill every demands at next placement $t = 3$ and period $t = 4$. Consequently, some disposables must be used. We

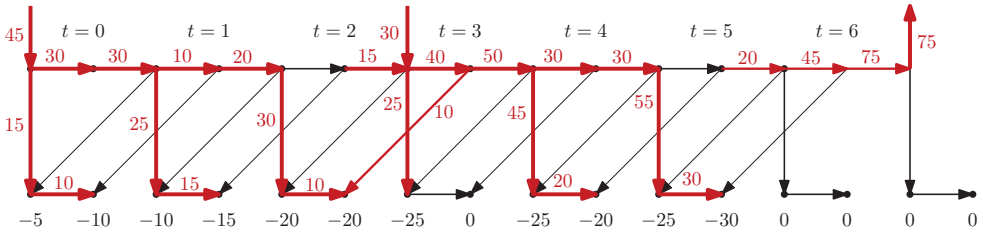
note that this example shows a very clear increasing demand trend with only one day off at time $(3, 1)$.

We can also create instances where Algorithm *Flow*.4.1 computes an optimal policy but not the algorithms from this Chapter. We present an example in Figure 5.9. In this example, the placements are periods 0 and 3, with the zero setup cost. However, the container price at period 3 is much higher than at period $t = 0$. Consequently, Algorithm *Flow*.5.1 will purchase more containers at period 0 to fulfill a demand at period $t = 2$ with relatively low disposable cost. However, because of a day off at period $t = 4$ and the close end of horizon, there is no reason to purchase containers at period $t = 3$ and letting them idle without fulfilling any demand. The additional containers purchased at period $t = 0$ only seemed profitable because we did not look further into the future.

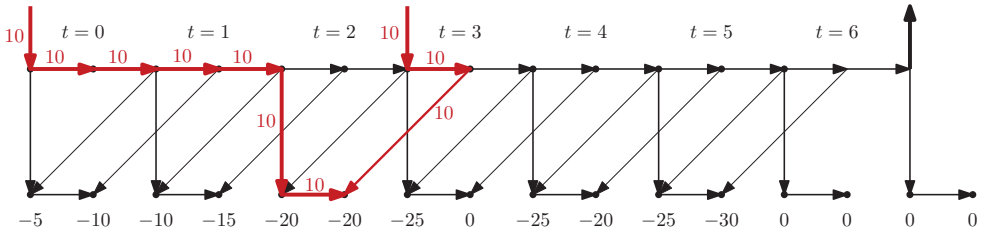
We conclude with two remarks on this last example. Firstly, if the time horizon contained one or two more periods with high demand, then Algorithm *Flow*.5.1 would have computed an optimal policy and Algorithm *Flow*.4.1 would have not. Secondly, we note that this issue is not easy to solve in



(a) Optimal flow, computed by Algorithm *Flow*.5.1

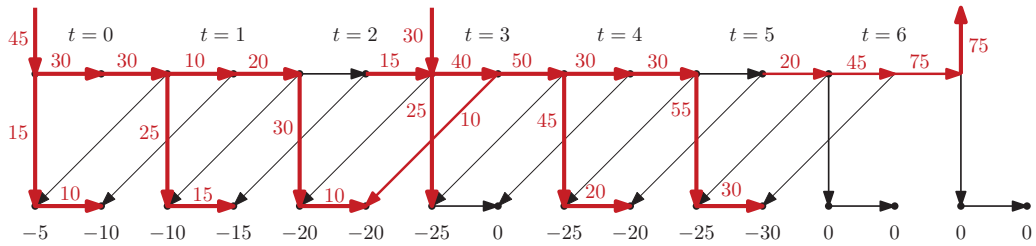


(b) Sub-optimal flow computed by Algorithm *Flow*.4.1

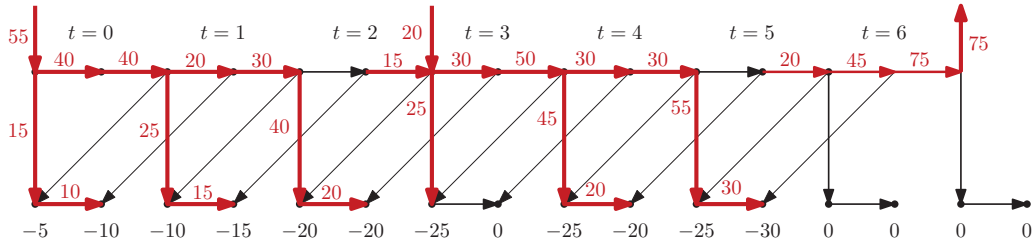


(c) Flow difference: a negative-cost cycle

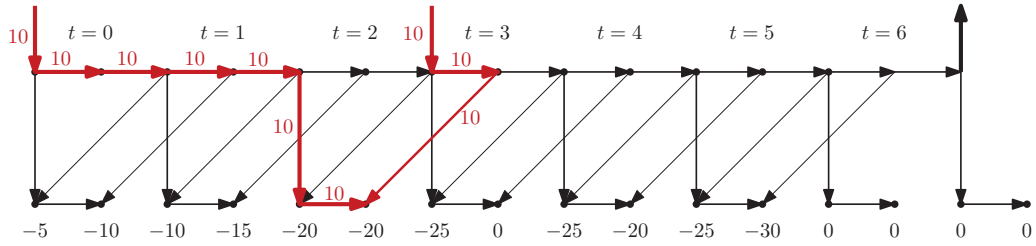
Figure 5.8: Example of a network where Algorithm *Flow*.5.1 computes an optimal policy but Algorithm *Flow*.4.1 does not. The flow quantities are given in red and the negatives excesses $-D(t, r)$ are in black.



(a) Optimal flow, computed by Algorithm *Flow*.4.1



(b) Sub-optimal flow computed by Algorithm *Flow*.5.1



(c) Flow difference: a negative-cost cycle

Figure 5.9: Example of network where Algorithm *Flow*.4.1 computes an optimal policy but Algorithm *Flow*.5.1 does not. The flow quantities are given in red and the demands values $-D(t,r)$ in black.

the general cases because whether it is profitable or not to purchase the additional containers at period 0 also depends on the future placements, especially when they have high container prices and some demand have a very low profitability.

5.5.2 Extensions

Using a similar reasoning as in Chapter 4, we can trivially extend Algorithms *Flow*.5.1 and *Flow*.5.2 to several suppliers. In a supply chain with I suppliers, the algorithms' complexity is then $O(T^2 \cdot \mathcal{M}cf_I)$, where $\mathcal{M}cf_I$ is the cost of a minimum linear-cost flow on a network $\hat{\mathcal{H}}(k_1, k_2)$ with $O(I \cdot R \cdot T)$ nodes and arcs.

If we assume that the distance between two placements is at least $\omega \geq 1$ periods, we can extend Algorithm *Flow*.5.1 to a delivery time between $\omega - 1$ and ω periods. If we do not set a minimum distance between two consecutive placements, we must compute 2^ω flows $\hat{f}_i$ instead of only $\hat{f}$ and $\hat{f}_2$.

Furthermore, our algorithms can easily be adapted to a bi-criteria problem, where the setup costs are replaced or completed with a second objective function of minimizing the number of placements. Then, the time complexity is $O(T^2 \cdot \mathcal{M}cf + T^3)$ for Algorithms 8 and 10, and $O(T^2 \cdot \mathcal{M}cf + T^4)$ for Algorithms $Flow$.5.1 and $Flow$.5.2. We believe that we cannot find any minimum linear-cost flow algorithm on the networks $\hat{\mathcal{H}}(k_1, k_2)$, so the time complexity should stay at $O(T^2 \cdot \mathcal{M}cf)$.

5.6 Network Analysis with Acyclic Flows

In this section, we develop an algorithm using the theorem of extreme points.

5.6.1 Introduction to Extreme Points

We recall that an acyclic flow contains by definition no cycle with positive flow on each arc. Moreover, the network $\mathcal{G}$ modeling the $DCPP$ has fixed-plus-linear cost, which is a special case of a network with concave cost. Theorem 3.2.5 states that there is an acyclic optimal flow in any concave-cost network. Given a flow and a period $k \in [0, T+1]$, we define the set $V_{\mathcal{G}}(k)$ as the connected component of node $v_{man}(k, 0)$ if k is a placement, and as an empty set otherwise. Consequently, there is an optimal flow so that the sets $\{V_{\mathcal{G}}(k)\}$ for $k \in [0, T+1]$ are disjoints (see Figure 5.10).

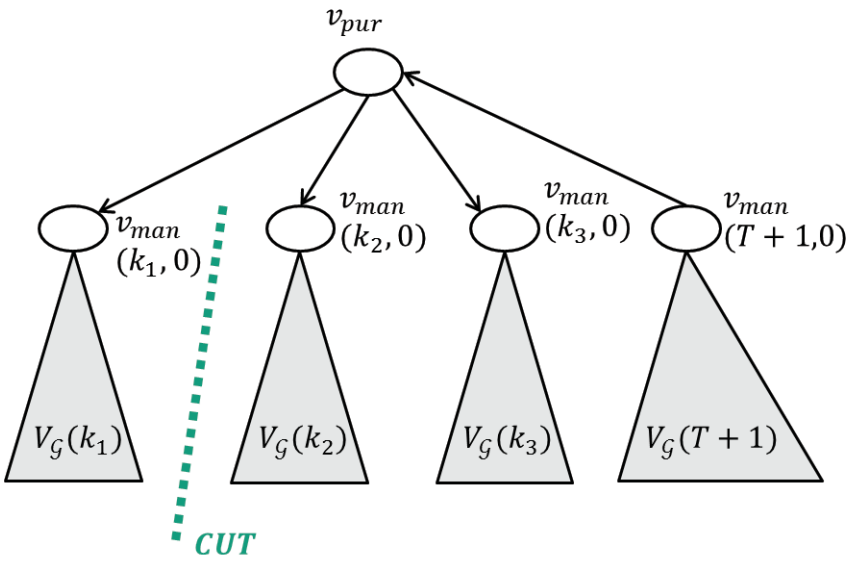


Figure 5.10: Tree representation of an acyclic flow on network $\mathcal{G}$.

In graph theory, a *cut* denotes a set of arcs separating the graph into two disconnected subgraphs. It this section, we enumerate several sets of cuts to compute a solution.

5.6.2 Cuts in the Wagner-Within algorithm

The dynamic lot-sizing problem can be modeled as illustrated in Figure 5.11. In this figure, we have $T = 5$, the leftmost node and the rightmost nodes correspond respectively to the state at the beginning of period 0 and at end of period 4.

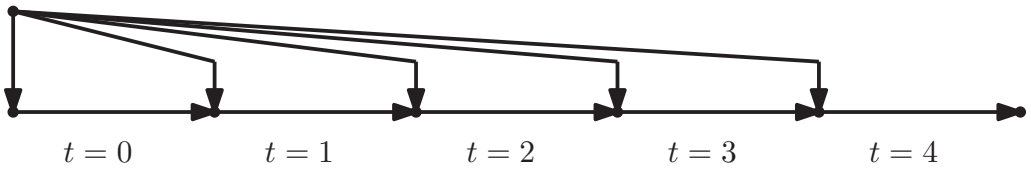


Figure 5.11: Network modeling of the dynamic lot-sizing problem.

Figure 5.12 formulates the zero-inventory property as a cut of the network, highlighting that the stock must be empty before purchasing. In this chapter, we extend this representation to our container management problem to compute an optimal policy.

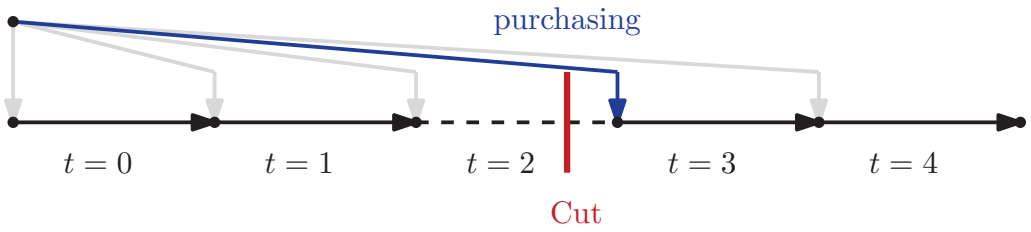


Figure 5.12: Network cut modeling the zero-inventory property. Non-considered purchasing arcs are marked in gray for better readability.

5.6.3 Streams

This subsection characterizes the connected components of the flow. For notational convenience, we write $a_{sup}(t, -1)$ to denote the arc $a_{ord}(0)$.

Arc Notations

Consider a flow f on a subnetwork of $\mathcal{G}$. We say that a disposable arc $a_{dis}(t, r)$ is *early* if r is an early time step and *late* otherwise. A non-disposable arc a is called *empty* if its flow $f(a)$ is zero and *partial* otherwise. We call a disposable arc $a_{dis}(t, r)$ *full* if $f\big(a_{sup}(t, r-1)\big) = 0$. We call it *partial* if its flow is positive but less than $D(t, r)$. A disposable arc which is neither full nor partial is called *empty*. For every positive demand $D(t, r)$, arc $a_{dis}(t, r)$ is:

1. empty if its flow is zero.

2. full if its flow is equal to $D(t, r)$.

3. partial otherwise.

We made the definitions more complex to extend the notations to arcs $a_{dis}(t, r)$ relative to an empty flow. We have then, for any flow and any time (t, r):

1. If $a_{dis}(t, r)$ is partial or empty, then $a_{dis}(t, r')$ is empty for all $r' \in]0, r[$.

2. If $a_{dis}(t, r)$ is partial or full, then $a_{dis}(t, r')$ is full for all $r' \in]r, R[$.

Moreover, any flow should have at most one partial arc per period.

Definition and Properties of Streams

Consider a flow f on $\mathcal{G}$. We denote by *stream* a connected component of partial arcs in the restriction of f to $\mathcal{G} - \{v_{pur}\}$. An example is given later in Figure 5.13. The following lemmas follow from the assumption that $L_{del} \leq R$. Therefore, at step $r = 0$, every container is either heading to the manufacturer or already there. Outgoing containers arrive to the manufacturer before the end of the period. We say that a flow *represents a policy* if it corresponds to the movement of containers in the process in an acyclic flow. We have shown in Chapter 3 every acyclic flow represents a policy under Hypothesis 2.2 [Cost] holds.

Lemma 5.6.1 *Consider a flow f in $\mathcal{G}$ representing a policy and a period t. Period t contains no partial arc if and only if no container is in the system.*

Proof:

On the one hand, if there is a partial arc, then there are containers in the system. On the other hand, if arcs $a_{ord}(t)$ and $a_{man}(t, R-1)$ are empty, then both the supplier and the manufacturer hold no container at every step of the period. Since $L_{del} \leq R$, every container outgoing at time $(t, 0)$ would have returned to the manufacturer, inducing a positive flow in $a_{man}(t, R-1)$. We conclude that there is no container in the system.

□

Lemma 5.6.2 *Consider a flow f in $\mathcal{G}$ representing a policy and a time (t, r). If arc $a_{sup}(t, r)$ is partial, then $a_{ord}(t)$ is partial and belongs to the same stream.*

Proof:

If arc $a_{sup}(t, r)$ is partial, then some containers are held by the supplier. These containers have been ordered at period t. Thus $a_{ord}(t)$ and every $a_{man}(t, r')$ for $r' \in [0, r]$ are partial and hence belong to the same stream.

□

Lemma 5.6.3 *Consider a flow f in $\mathcal{G}$ representing a policy and a period t. If arc $a_{man}(t,r)$ is partial, then $a_{man}(t,R-1)$ is partial and belongs to the same stream.*

Proof:

If arc $a_{man}(t,r)$ is partial, then some containers are held by the manufacturer at time (t,r). These containers stay in the manufacturer stock at least until time $(t,R-1)$. Consequently, every arc $a_{man}(t,r')$ with $r' \in [r,R[$ is partial and hence belong to the same stream.

□

Proposition 5.6.4 *Consider a flow f representing a policy and a period t. The partial arcs from period t come from at most two streams. Moreover, if there are two streams, then*

1. *one stream contains arc $a_{man}(t,R-1)$,*
2. *the other stream contains arc $a_{ord}(t)$.*
3. *Arc $a_{man}(t,0)$ is empty,*
4. *No early disposable arc $a_{dis}(t,r)$ is partial, for $r \in [0,R-L_{del}]$.*

Proof:

Lemmas 5.6.2 and 5.6.3 show that every non-disposable arc at period t belongs to a stream including either $a_{ord}(t)$ or $a_{man}(t,R-1)$. If a disposable arc $a_{dis}(t,r)$ is partial, then arc $a_{sup}(t,r-1)$ must be partial, so $a_{dis}(t,r)$ belongs to the same stream as $a_{ord}(t)$.
Furthermore, if $a_{man}(t,0)$ is partial, then a container is idle and the two streams are connected, which is a contradiction. If there is an early disposable arc $a_{dis}(t,r)$, then a container is hold by the supplier from time $(t,0)$ to time (t,r) and by the manufacturer from time $(t,r+L_{ord})$ to time $(t+1)$. Thus the two arcs $a_{ord}(t)$ and $a_{man}(t,R-1)$ are connected, which is a contradiction.

□

When there are two stream at a period t, we denote by *supplier-stream* the stream including arc $a_{ord}(t)$ and by *manufacturer-stream* the one including arc $a_{man}(t,R-1)$. We say that period t *contains* a stream $\mathcal{S}tr$, or equivalently that stream $\mathcal{S}tr$ *is in period* t, if $\mathcal{S}tr$ contains arc $a_{ord}(t)$ or $a_{man}(t,R-1)$.
Given a flow, we call the *start* (respectively the *end*) of a stream the first (respectively the last) period it is in. By Hypothesis 2.1, the arcs are either between two nodes from a same period or between nodes of consecutive periods. Therefore, any stream is in any period between its start and its end. A stream containing arc $a_{ord}(t)$ also contains node $v_{man}(t,0)$ and a stream

containing arc $a_{man}(t, R-1)$ also contains node $v_{man}(t+1, 0)$. Therefore, a stream containing node $v_{man}(t, 0)$ is in both periods t and $t+1$.

Proposition 5.6.5 *Consider a flow f representing a policy and a period t. If there are two streams over periods t and $t+1$, then there exists $r \in]R - L_{del}, R[$ so that $a_{dis}(t, r)$ is partial.*

Proof:

Among the two streams, one stream $\mathcal{S}tr_1$ contains node $v_{man}(t+1, 0)$ and the other stream $\mathcal{S}tr_2$ contains nodes $v_{man}(t, 0)$ and $v_{man}(t+2, 0)$. Consequently, nodes $v_{man}(t, 0)$ and $v_{man}(t+2, 0)$ are connected by a path of partial arcs. This path contains either $a_{man}(t, R-1)$ or $a_{dis}(t, r)$ for some $r \in [R - L_{del}, R[$. Since $v_{man}(t+1, 0)$ and $v_{man}(t, 0)$ are on different streams, the stream including $v_{man}(t, 0)$ cannot contain arc $a_{man}(t, R-1)$ or $a_{dis}(t, R - L_{del})$. Therefore it must contain a late disposable arc $a_{dis}(t, r)$, for some $r \in]R - L_{del}, R[$.

□

Corollary 5.6.6 *Consider a flow f representing a policy and a period t so that two streams are both in periods t and $t+1$ Then each of the streams is the manufacturer-stream of one of the periods and the supplier-stream of the other period.*

5.6.4 Cuts

Introduction

A difficulty to decompose the flow is that a cut must separate the flow on two echelons, namely on the manufacturer echelon as well as on the supplier echelon. In particular, Figure 5.13 presents a possible decomposition of optimal solution into streams. Suppose that the demand only increases from an even period to the next one but is constant over all odd periods. Then an optimal policy may only purchase containers at even periods. The associated optimal flow contains a single stream (in green) for the demand at odd periods and one stream (in gray) per even period with purchasing. In this case, we have no simple cut of network $\mathcal{G} - \{v_{pur}\}$ between a subgraph for early periods and a subgraph for late periods.

Start-Cuts

Consider a flow f in $\mathcal{G}$, a period $t_s \in [-1, T+2]$ and a time step $r_s \in [L_{del} - 1, R[$. We say that the flow contains the *start-cut* $\mathcal{C}ut_s(t_s, r_s)$ when (see Figure 5.14):

1. Arc $a_{man}(t_s, r_s)$ is empty.

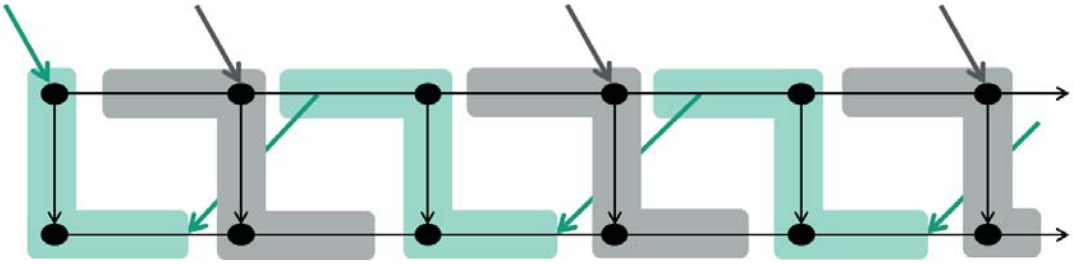


Figure 5.13: Flow decomposition with a long stream (in green) starting before and ending after several smaller streams (in gray).

2. If $r_s < R - 1$, then arc $a_{man}(t_s, r_s + 1)$ is not empty.
3. $\forall r \in [0, R - L_{del}]$, arc $a_{dis}(t_s, r)$ is not partial.

Note that the second condition ensures that a flow does not contain two start-cuts at the same period.

Lemma 5.6.7 *Any flow f representing a policy and containing the start-cut $\mathcal{C}ut_s(t_s, r_s)$ is such that:*

- *For each $r \in [0, r_s]$, arc $a_{man}(t_s, r)$ is empty.*
- *For each $r \in [1, r_s]$, arc $a_{dis}(t_s, r - L_{del})$ is full.*
- *Arc $a_{dis}(r_s + 1 - L_{del})$ is empty and demand $D(r_s + 1 - L_{del})$ is positive.*

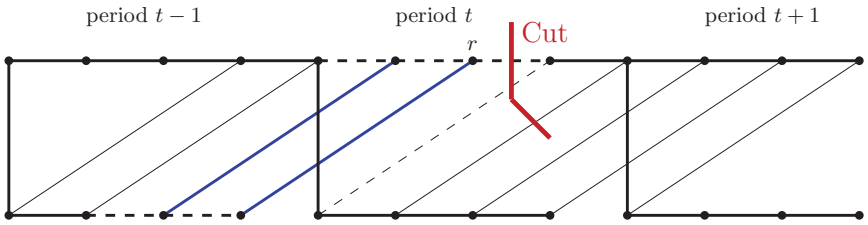


Figure 5.14: Start-cut $\mathcal{C}ut_s(t, r)$ (in red), crossing non-partial arcs. Blue arcs must be full and dashed arcs must be empty.

Proposition 5.6.8 *Consider a flow f representing a policy. If a stream starts at period t_s, then f contains a start-cut at period t_s.*

Proof:

Suppose that a stream $\mathcal{S}tr$ starts at period t_s. It contains $v_{man}(t_s + 1, 0)$, but not $v_{man}(t_s, 0)$ since the stream is not in period $t_s - 1$. Arc $a_{man}(t_s, 0)$ must be empty and every early disposable arc $a_{dis}(t_s, r)$ must not be partial, as otherwise there would be a path from $v_{man}(t_s, 0)$ to $v_{man}(t_s + 1, 0)$, hence

$v_{man}(t_s, 0)$ would belong to the stream. For every late demand $D(t_s - 1, r)$, arc $a_{dis}(t_s, r)$ is not partial, as otherwise the stream would be in period $t_s - 1$. We consider the last empty arc $a_{man}(t_s, r_s)$. No arc $a_{dis}(t_s, r)$ with $r \in [0, R - L_{del}]$ can be partial, as otherwise there would be a path of partial arcs from $v_{man}(t_s, 0)$ to $v_{man}(t_s + 1, 0)$ and the stream would be at period $t_s - 1$. We conclude that there is a start-cut at period t_s.

□

Remark 5.6.9 *Consider a flow on $\mathcal{G} - v_{pur}$ and let $\mathcal{G}_t$ be the restriction of this flow to time $(t + 1, 0)$. A start-cut at period t is a cut of the flow on $\mathcal{G}_t$ such that one of the connected components is in periods t and $t + 1$. However, a start-cut does not necessarily induces a new stream on $\mathcal{G} - v_{pur}$, because the two connected components on $\mathcal{G}_t$ may be connected after time t.*

End-Cuts

Consider a flow f in $\mathcal{G}$, a period $t_e \in [-1, T + 2]$ and a time step $r_e \in [0, R]$. We say that the flow contains the *end-cut* $\mathcal{C}ut_e(t_e, r_e)$ when (see Figure 5.15):

1. Arc $a_{man}(t_e, 0)$ is empty.
2. $\forall r \in [0, r_e[$, arc $a_{dis}(t_e, r)$ is empty.
3. $\forall r \in [r_e, R[$, arc $a_{dis}(t_e, r)$ is full.

Similar to start-cuts, the second condition ensures that there is at most one end-cut per period. Note that the set $[0, r_e[$ is empty for $r_e = 0$, whereas the set $]r_e, R[$ is empty for $r_e = R$.

Lemma 5.6.10 *Any flow f representing a policy and containing the end-cut $\mathcal{C}ut_e(t_e, r_e)$ is such that:*

- *If $r_e > 0$ then $a_{sup}(t_e, r_e - 1)$ is empty. Otherwise $a_{ord}(t_e)$ is empty.*
- *If $r_e > 2$, then $a_{sup}(t_e, r_e - 2)$ is not empty. If $r_e > 1$, then $a_{ord}(t_e)$ is not empty.*
- *For each $r \in [r_e, R[$, arc $a_{sup}(t_e, r)$ is empty.*

Proposition 5.6.11 *Consider a flow f on $\mathcal{G}$. If a stream ends at period t_e, then f contains an end-cut at period t_e.*

Proof:

Suppose that a stream ends in period t_e. It contains $v_{man}(t_e, 0)$, but not $v_{man}(t_e + 1, 0)$ since it is not in period $t_e + 1$. Arc $a_{man}(t_e, 0)$ must be empty and every early disposable arc $a_{dis}(t_e, r)$ must not be partial, as otherwise

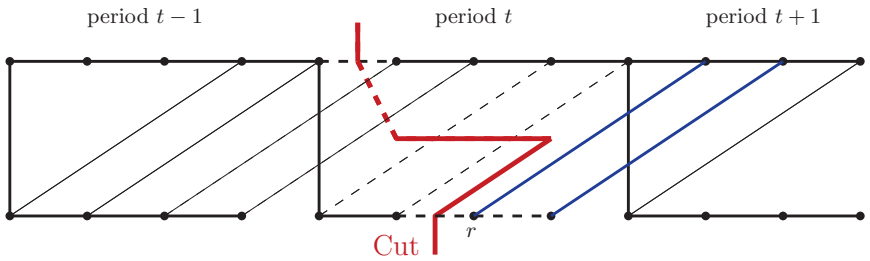


Figure 5.15: End-cut $\mathcal{C}ut_e(t, r)$ (in red), crossing non-partial arcs. Blue arcs must be full and dashed arcs must be empty.

there would be a path from $v_{man}(t_e, 0)$ to $v_{man}(t_e+1, 0)$, hence $v_{man}(t_e+1, 0)$ would belong to the stream. If every disposable arc $a_{dis}(t_e, r)$ with $r \in [0, R[$ is empty, then we have the end-cut $\mathcal{C}ut_e(t_e, R)$. Otherwise, consider the first full disposable arc $a_{dis}(t_e, r_e)$ in period t_e. Every arc $a_{dis}(t_e, r)$ with $r < r_e$ is empty. In addition, arc $a_{sup}(t_e, r_e)$ is empty so there is no container in the supplier stock after time (t_e, r_e). Thus every disposable arc after time step r_e is full and we have the end-cut $\mathcal{C}ut_e(t_e, r_e)$.

□

Remark 5.6.12 *Similar to start-cuts, there may be an end-cut at period t without any stream ending. This occurs when there is a single stream at period $t-1$.*

Middle-Cuts

Consider a flow f in $\mathcal{G}$, and a period $t_m \in [-1, T+2]$. We say that this flow contains the *middle-cut* $\mathcal{C}ut_m(t_m)$ when (see Figure 5.16):

1. Arc $a_{man}(t_m, 0)$ is empty.
2. $\forall r \in [0, R - L_{del}]$, arc $a_{dis}(t_m, r)$ is empty.

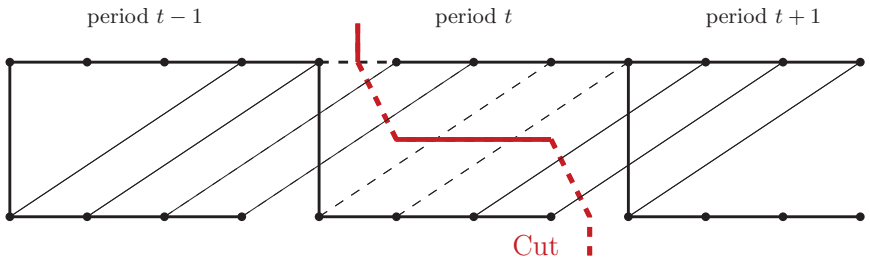


Figure 5.16: Example of a middle-cut (in red). Dashed arcs (in black) must be empty.

Proposition 5.6.13 *Consider a flow f on $\mathcal{G}$. If two streams are in periods $t_m - 1$, t_m and $t_m + 1$, then f contains a middle-cut at period t_m.*

Proof:

Necessarily, one stream contains nodes $v_{man}(t_m - 1, 0)$ and $v_{man}(t_m - 1, 0)$ while the other stream contains node $v_{man}(t_m, 0)$ and $v_{man}(t_m + 2, 0)$. We call $\mathcal{S}tr_1$ the former stream and $\mathcal{S}tr_2$ the later one. By Proposition 5.6.4, arc $a_{man}(t, r)$ is empty and every early arc $a_{dis}(t_m, r)$ is not partial. Moreover, by Proposition 5.6.5 there is a partial arc $a_{dis}(t_m, r_m)$ for some $r_m \in]R - L_{del}, R[$. Therefore, every early disposable arc $a_{dis}(t_m, r)$ with $r \leq R - L_{del} < r_m$ must be empty. We have thus the middle-cut $\mathcal{C}ut_m(t_m)$.

□

5.6.5 Extreme Points Theorem

Lemma 5.6.14 *Consider a flow in $\mathcal{G}$ and a positive demand $D(t, r)$. Arc $a_{dis}(t, r)$ is full if and only if $v_{sup}(t, r)$ is not adjacent to any partial arc.*

Proof:

Node $v_{sup}(t, r)$ is incident to at most three arcs: $a_{dis}(t, r)$, $a_{sup}(t, r)$ and either $a_{sup}(t, r-1)$ or $a_{ord}(t)$. Arc $a_{dis}(t, r)$ is full if and only if $a_{sup}(t, r-1)$ is empty. Then, arc $a_{sup}(t, r)$ must be empty for the imbalance inequality to hold.

□

The following lemma adapts Theorem 3.2.5 to our network.

Lemma 5.6.15 *There is an optimal flow on $\mathcal{G}$ such that every induced stream in $\mathcal{G}$ contains at most one manufacturer node $v_{man}(k, 0)$ where k is a placement. Reciprocally, if k is a placement, then $a_{pur}(k)$ is adjacent to a partial arc and hence to a stream.*

Proof:

Theorem 3.2.5 states that there exists an acyclic optimal flow. If a stream in this flow contained two purchasing arcs $a_{pur}(k_1, 0)$ from v_{pur} to $v_{man}(k_1, 0)$ and $a_{pur}(k_2, 0)$ from v_{pur} to $v_{man}(k_2, 0)$ with $k_1 \neq k_2$, there would be a cycle in the flow.

The destination node $v_{man}(k, 0)$ of arc $a_{pur}(k)$ has excess $D(k-1, R - L_{del})$ and is incident to four other arcs: $a_{end}(k)$, $a_{dis}(k - 1, R - L_{del})$, $a_{man}(k, 0)$ and $a_{ord}(k)$ Arc $a_{end}(k)$ is empty because arc $a_{pur}(k)$ is not[1]. The flow in $a_{dis}(k-1, R-L_{del})$ is at most $D(k-1, R-L_{del})$, so at least $f\big(a_{pur}(k)\big) > 0$ flow units go through arcs $a_{man}(k, 0)$ and $a_{ord}(k)$ combined. Thus, at least one of these non-disposable arcs has a positive flow and therefore is partial.

□

By Lemma 5.6.15, if period k is a placement, then $v_{man}(k, 0)$ belongs to a stream and we say that this stream *contains* placement k. The following lemma states that we do not need the knowledge of full arcs to build a flow.

[1]Recall that these two arcs are incident to the same two nodes.

Lemma 5.6.16 *Any flow f on network $\mathcal{G}$ is fully characterized by its purchasing sizes and the flow in its streams.*

Proof:

Any flow f is defined by its value in arcs with positive flow, i.e. in partial arcs and full arcs. By Lemma 5.6.14, every supplier node $v_{sup}(t, r)$ incident to a full arc in f is not incident to any partial arc. Therefore, given the flow in the partial arcs we can reconstruct the flow f by correcting the imbalance property of every supplier node via a disposable arc.

□

Remark 5.6.17 *Consider a stream which is part of a flow on $\mathcal{G}$. Then the total excess of its nodes is negative if and only if it is a purchasing stream. The total excess of its nodes cannot be positive.*

5.6.6 Stream-based Algorithm

We define a *block* of a stream as an interval of periods in which this stream is the only one, and a *branch* of a stream as an interval of periods on which there is a second stream. We say that two streams $\mathcal{S}tr_1$ and $\mathcal{S}tr_2$ are *adjacent* if there is a period containing both streams. A stream adjacent to a partial purchasing arc is called *purchasing-stream*. When decomposing a flow into streams, we deduce from the total excess in each stream whether this stream is a purchasing stream or not, by Remark 5.6.17.
We denote by *branch-stream* a stream containing no block and a *block-stream* a stream containing at least one block. Any branch of any stream alternates over time between being the supplier-stream, and the manufacturer stream. We consider in particular a stream with a single block, which we call *one-block-stream*. A one block stream is composed of block and up to two branches, one in earlier periods and the other in later periods. The following proposition is the central result to construct our acyclic flow using dynamic programming

Proposition 5.6.18 *A one-block-stream is characterized by two start-cuts $\mathcal{C}ut_s(t_1, r_1)$, $\mathcal{C}ut_s(t_3, r_3)$ and two end-cuts $\mathcal{C}ut_e(t_2, r_2)$, $\mathcal{C}ut_s(t_4, r_4)$ where $t_1 \leq t_2 < t_3 \leq t_4$ and such that $t_2 - t_1$ and $t_4 - t_3$ are even numbers.*

Proof:

This proposition follows from the propositions on the cuts. We refer to Figure 5.17 for an illustration. By Proposition 5.6.8, every stream starts with a start-cut and by Proposition 5.6.11 every stream ends with an end-cut. Consider a stream with a single block. This stream begins with a start-cut $\mathcal{C}ut_s(t_1, r_1)$ and ends with an end-cut $\mathcal{C}ut_e(t_4, r_4)$. Consider the earliest period $k_2 + 1 > k_1$ of the block from this stream. There are two streams at

period k_2, thus there is a cut $\mathcal{C}ut_e(t_2, r_2)$ ending this second stream. Consider the last period $t_3 - 1 < t_4$ of the block. A new stream starts at period t_3 so there is a start-cut $\mathcal{C}ut_s(t_3, r_3)$. In every period between t_1 and $t_2 - 1$ and between $t_3 + 1$ and t_4, there are two streams. By Proposition 5.6.13, there is thus a middle-cut at every of these periods. By definition of the cuts and by Lemmas 5.6.7 and 5.6.10, each cut at a period t with two streams defines which arc belongs to which stream at that period. Therefore, the four cuts $\mathcal{C}ut_s(t_1, r_1)$, $\mathcal{C}ut_e(t_2, r_2)$, $\mathcal{C}ut_s(t_3, r_3)$, and $\mathcal{C}ut_s(t_4, r_4)$ characterize the stream.
Finally, by Corollary 5.6.6, the one-block-stream alternates between being a supplier-stream and a manufacturer-stream. Since the one-block-stream is the manufacturer-stream of periods t_1 and t_2 (respectively t_3 and t_4), $t_2 - t_1$ (respectively $t_4 - t_3$) needs to be a even number.

□

Proposition 5.6.19 *A branch-stream is defined by a start-cut $\mathcal{C}ut_s(t_1, r_1)$ and an end-cut $\mathcal{C}ut_e(t_4, r_4)$ such that $t_1 < t_2$ and $t_4 - t_1$ is an odd number.*

Proof:

By Proposition 5.6.8, a branch-stream starts with a start-cut $\mathcal{C}ut_s(t_1, r_1)$ and ends with an end-cut $\mathcal{C}ut_e(t_4, r_4)$ so that $t_1 \leq t_4$. A branch-stream starts as a manufacturer-stream, ends as a supplier-stream and alternates between being the supplier-stream and the manufacturer-stream. Therefore, $t_4 - t_1$ is an odd number.

□

Remark 5.6.20 *Using the notations of Proposition 5.6.18, a branch-stream is the extension of a block-stream to $t_3 = t_2 - 1$, where the branches in period intervals $[t_1, t_2]$ and $[t_3, t_4]$ are connected.*

Using the notations from Proposition 5.6.18, we note that between periods t_1 and t_2 (respectively periods t_3 and t_4), there may be several start-cuts and end-cuts delimiting the flow between several branch-streams. All these branch-streams would be adjacent to the one-block-stream, but these start-cuts and end-cuts have no influence on which arc is in the one-block-stream.

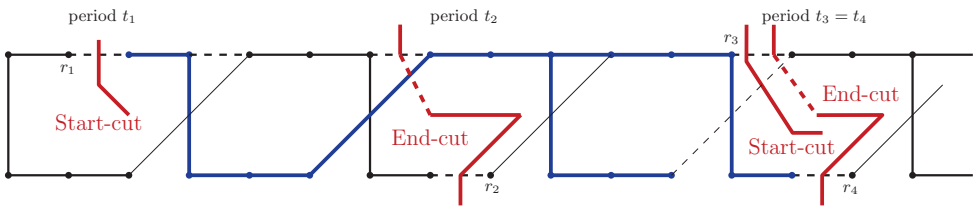


Figure 5.17: A stream with a single block (in blue) is defined by up to two start-cuts and two end-cuts (in red).

We show that we can compute these streams together with a single minimum linear-cost flow.

We say that an arc is *adjacent* to a stream if it is adjacent to a node in this stream. Consider two adjacent streams. By definition, at one period t one of these streams is the supplier-stream and the other is the manufacturer-stream. By Proposition 5.6.4, every early disposable arc at period t is empty and hence is adjacent to both of these streams. Adding these disposable arcs to the partial arcs in the two streams build a connected component.

We call *pack of streams* a set of several adjacent streams and some non-partial arcs building together a purchasing-stream. Any pack of stream can therefore be generated from a purchasing-stream by removing some partial arcs. The cost of a minimum linear-cost flow on the subnetwork delimited by these arcs is lower or equal to the total cost of minimum linear-cost flows computed of each of the streams and non-partial arcs from the pack. Consequently, we do not need to consider every stream decomposition to compute a minimum cost flow on $\mathcal{G}$. Instead, we can only consider every decomposition in packs of streams. We refer to a pack of streams as a *one-block-pack* if the corresponding partial arcs build a single block, as illustrated on Figure 5.18. With the same arguments as in Proposition 5.6.18, a one-block-pack is also characterized by two start-cuts and two end-cuts.

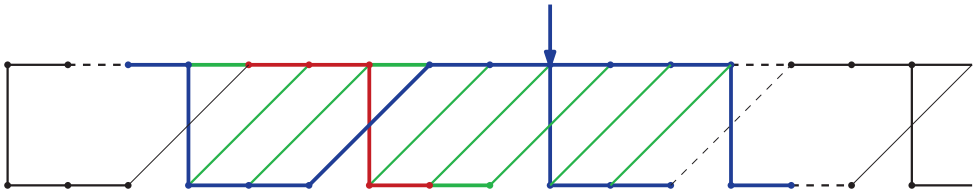


Figure 5.18: A one-block-pack composed of a one-block-stream (in blue), a branch-stream (in red) and several non-partial arcs (in green). Note that if the green supplier arc was not in the one-block-pack, the pack of streams would not delimit a one-block-stream.

We denote by $h^*(t_1, r_1, t_2, r_2, t_3, r_3, t_4, r_4, k)$ a locally optimal flow on the one-block-pack delimited by the start-cuts $\mathcal{C}ut_s(t_1, r_1)$, $\mathcal{C}ut_s(t_3, r_3)$, the end-cuts $\mathcal{C}ut_s(t_2, r_2)$, $\mathcal{C}ut_e(t_4, r_4)$ and with a placement at period $k \in [t_1 + 1, t_4]$. We denote by $h^*(t_1, r_1, t_2, r_2, t_3, r_3, t_4, r_4)$ a best of these flows:

$$H^*(t_1, r_1, t_2, r_2, t_3, r_3, t_4, r_4) := \min_{k \in [t_1+1, t_4]} \left\{ H^*(t_1, r_1, t_2, r_2, t_3, r_3, t_4, r_4, k) \right\} \tag{5.21}$$

We denote by $f^*(t_3, r_3, t_4, r_4)$ the best flow from time $(0, 0)$ delimited by the start-cut $\mathcal{C}ut_s(t_3, r_3)$ and the end-cut $\mathcal{C}ut_s(t_4, r_4)$. Algorithm 12 computes $f^*(t_3, r_3, t_4, r_4)$ as the best combination of flows $f^*(t_1, r_1, t_2, r_2)$ and $h^*(t_1, r_1, t_2, r_2, t_3, r_3, t_4, r_4)$ over every possible previous start-cut $\mathcal{C}ut_s(t_1, r_1)$ and end-cuts $\mathcal{C}ut_e(t_2, r_2)$.

Algorithm 12: Stream-based algorithm

Set $f^*(0, r_1, 0, r_2) = 0$ for all r_1, r_2;
for *End-cut* $\mathcal{C}ut_e(t_4, r_4)$ *in increasing sequence* **do**
  for *Start-cut* $\mathcal{C}ut_s(t_3, r_3)$ *in increasing sequence* **do**
    foreach *End-cut* $\mathcal{C}ut_e(t_2, r_2)$ *with* $t_2 \leq t_3 + 1$ **do**
      foreach *Start-cut* $\mathcal{C}ut_s(t_1, r_1)$ *with* $t_1 \leq t_2, t_3$ **do**
        foreach *Placement* k *in* $[t_1 + 1, t_4]$ **do**
          Compute $h^*(t_1, r_1, t_2, r_2, t_3, r_3, t_4, r_4, k)$with a minimum linear-cost flow;
        Deduce $h^*(t_1, r_1, t_2, r_2, t_3, r_3, t_4, r_4)$;
    Deduce $f^*(t_3, r_3, t_4, r_4)$ minimizing the cost $F^*(t_1, r_1, t_2, r_2) + h^*(t_1, r_1, t_2, r_2, t_3, r_3, t_4, r_4)$;
return $f^*(T + 1, 0, T + 1, 0)$;

Theorem 5.6.21 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. If there exists an acyclic minimum cost flow such that every stream contains at most one block, then Algorithm 12 computes and optimal solution to DCPP in* $O(T^5 \cdot R^4 \cdot \mathcal{M}cf)$ *time, where* $\mathcal{M}cf$ *denotes the complexity of a minimum linear-cost flow.*

Proof:

The computation of every locally optimal flows $h^*(t_1, r_1, t_2, r_2, t_3, r_3, t_4, r_4, k)$ takes $O(T^5 \cdot R^4 \cdot \mathcal{M}cf)$ time. The algorithm then iterates over two start-cuts and two end-cuts, which takes $O(T^4 \cdot R^4)$ time. Therefore, the time complexity of Algorithm 12 is $O(T^5 \cdot R^4 \cdot \mathcal{M}cf)$.
By Propositions 5.6.18 and 5.6.19, the algorithm iterates over every one-block-stream and every branch-stream. Since, by assumption one of these combinations builds an optimal flow, we conclude that the algorithm computes an acyclic minimum cost flow.

□

Theorem 5.6.22 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. If there exists an acyclic minimum cost flow such that every purchasing-stream contains at least one block, then Algorithm 12 computes and optimal solution to DCPP in* $O(T^5 \cdot R^4 \cdot \mathcal{M}cf)$ *time, where* $\mathcal{M}cf$ *denotes the complexity of a minimum linear-cost flow.*

Proof:

By Proposition 5.6.18 and by definition of a one-block-pack, the algorithm iterates over every possible one-block-pack. We show that the acyclic optimal flow can be decomposed into one-block-packs.

Consider a purchasing-stream. If it contains several blocks, then by assumption every branch stream between two blocks contains no placement. We can thus generate a one-block-pack from this purchasing stream by adding these branch-streams. It follows that the acyclic optimal flow can be decomposed into one-block-packs and thus the Algorithm 12 computes an optimal solution to $DCPP$.

□

Proposition 5.6.23 *Suppose that Hypotheses 2.1 [Delay], 2.2 [Cost], and 5.2 [Idleness] hold. Then Algorithm 12 computes an optimal solution to $DCPP$.*

Proof:

We prove that under the hypotheses, every purchasing-stream has at least one block. Consider a placement k. By Hypothesis 5.2 [Idleness], arc $a_{man}(k, 0)$ is partial. Thus, by Proposition 5.6.4 there is a single stream at period k. Therefore this stream is a purchasing-stream with at least one block.

□

It follows from Theorem 5.6.21 that if we the number of blocks is limited to a constant value, then the problem is still polynomial.

Theorem 5.6.24 *if Hypothesis 2.1 [Delay] and 2.2 [Cost] hold and if every stream contains at most K blocks, then a minimum cost flow on $\mathcal{G}$ can be computed in polynomial time in T and R and factorial time in K.*

Proof:

Each block-stream $\mathcal{S}tr$ is defined by its start-cut, the end of a previous stream, the start of a future stream, its end-cut, its placement as well as the placement of every branch-stream between two of its blocks. There are at most K blocks, so there are at most K placements to consider: one for $\mathcal{S}tr$, and one for each of the $K - 1$ branch-streams. We thus compute the minimum cost flow in $\mathcal{S}tr$ and every potential branch-stream between two of its blocks by computing $K!$ minimum linear-cost flows.

□

Proposition 5.6.25 *If Hypotheses 2.1 [Delay], 2.2 [Cost], 4.1 [Demand], and 4.2 [Placement] hold, then Algorithm 12 computes an optimal solution to $DCPP$.*

Proof:

Consider a placement k. By assumption, no disposable is bought at periods k and $k + 1$. If a container is idle at period k then the purchasing-stream

contains at least one block. If no container is idle at period k but no disposable is bought at period $k-1$, then there is a cut at period k ending every stream. Therefore, the stream $\mathcal{S}tr_k$ with placement at period k is not inside of a block of another purchasing stream.
Suppose finally that a disposable is bought for a late demand at period $k-1$ and that no container is idle at period k. Because the demand is non-decreasing, the containers arriving to the manufacturer from the late demands of period $k-1$ are not sufficient to fulfill the demand from period $k+1$. We deduce that some containers are purchased at period $k+1$. Moreover, the stream $\mathcal{S}tr_{k+1}$ with placement at period $k+1$ is the manufacturer-stream at period k. Consequently, these two streams cannot be inside of a block from another purchasing-stream.
From these three cases, it follows that during a block of a purchasing-stream there is at most one purchasing-stream and this stream (if any) would be composed of a single branch. We conclude by noting that any stream without placement can be included to a purchasing stream into a bigger virtual purchasing-stream.

□

5.6.7 Limitation of the Stream-based Approach

It appears with Theorem 5.6.24 that a stream composed of several blocks requires a high computational effort, since we may have branch-streams with placement between two blocks.
Figure 5.19 describes a flow difficult to construct using a stream-based approach. In the example, the red arcs represent a minimum acyclic cost flow. We can choose any cost function such that this flow is optimal.
The optimal flow is composed of two purchasing streams, namely a branch-stream in periods $t \in [2,3]$ and a block-stream composed of two blocks in periods $[0,2]$ and $[3,4]$. Algorithm 12 disregard purchasing-streams composed a single branch between two block of another purchasing stream.
We have shown in Theorem 5.6.24 that a stream-based algorithm similar to Algorithm 12 can still compute an optimal acyclic flow in polynomial time. However, we can build flow networks where every optimal acyclic flow contains a stream with $O(T)$ blocks such that there is a placement in each branch-stream between every block. The number of possible decompositions in stream is factorial in T, which leads us to the following conjecture:

Conjecture 5.6.26 *DCPP is NP-hard in the general case.*

Finally, note that the optimal flow fulfills every demand during the first two periods after each placement. Therefore, Algorithm *Flow*.4.1 computes an optimal solution and Algorithm 12 does not, even though both algorithms are optimal under the hypotheses from Chapter 4.

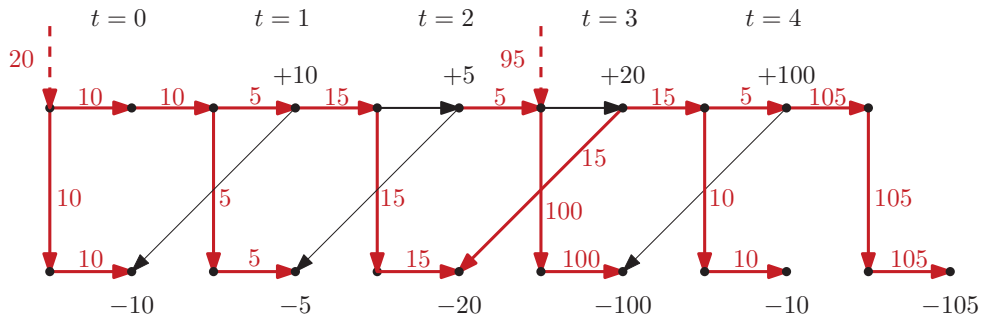


Figure 5.19: Example of acyclic optimal flow (in red) which cannot be generatd using Algorithm 12. The positive flow is denoted in red, whereas the excess values are in black. There is no early demand.

5.7 Outlook

This chapter analyzes the *DCPP* under a general demand pattern. We firstly present optimal polynomial-time algorithms in the special cases of zero delay and of zero disposables. The algorithm forbidding disposables is an extension of an algorithm from Chapter 4.

We then propose two polynomial-time algorithms *Flow*.5.1 and *Flow*.5.2 using the same framework as Algorithms *Flow*.4.1 and *Flow*.4.2 from the previous chapter. Algorithm *Flow*.5.1 is optimal whenever there exists an optimal solution letting an idle container at every placement. Algorithm *Flow*.5.2 extends it to allow close placements, such that Hypothesis 5.2 [Idleness] only needs to hold for placements which are not followed by another placement at the next period. In addition, we provide a certificate of optimality of Algorithms *Flow*.5.1 and *Flow*.5.2. Consequently, the algorithms may ensure that the generated policy is optimal.

All four flow algorithms *Flow*.4.1, *Flow*.4.2, *Flow*.5.1 and *Flow*.5.2 have the same complexity bound $O(T^2 \cdot \mathcal{M}cf)$. Moreover, we provide examples showing that none of *Flow*.4.1 and *Flow*.5.1 is strictly better than the other.

Finally, we propose a general algorithmic framework to compute efficient solutions. We conjecture that the *DCPP* is NP-hard and give a class of flow networks such that an optimal policy can be computed in polynomial time.

Chapter 6

Computational Study

This chapter focuses on experimental performances of our algorithms, bence both their execution speed and the quality of the generated policies. In Chapters 4 and 5, we presented several algorithms to solve $DCPP$. On the one hand, there are four[1] *flow-based algorithm* computing several minimum linear-cost flows, namely $Flow$.4.1, $Flow$.4.2, $Flow$.5.1 and $Flow$.5.2. These algorithms are the result of our study and we will show that they are efficient. On the other hand, in Chapter 5 we described two algorithms, one computing an optimal policy when disposables were not allowed and then other when transportation delays are null.
The four flow algorithms represent our solutions to $DCPP$, whereas the last two algorithms are used as a reference to measure the performance of the flow algorithms and the impact of lead times and disposables on the optimal policy.
In the first section, we compare the performance of the different algorithms. In Sections 6.2 to 6.4, we propose several ways of improving the running time of our flow algorithms. Our three improvements include multiple threading, an alternative network formulation and an alternative algorithm to compute minimum linear-cost flows. Simulations for the last two improvements are presented together.

6.1 Algorithms Performance

In Chapters 4 and 5, we propose several algorithms to solve $DCPP$:

- Algorithm $Ndis.D$ computes the best policy without disposable and with minimum cost. Each chapter proposes a version, one for non-decreasing demand and one for general demand.

[1] In Chapter 5, we have presented two other algorithms 8 and 10 as preliminary algorithms to Algorithms $Flow$.5.1 and $Flow$.5.2. These two algorithms are disregarded in this chapter.

- Algorithm $NDel.D$ computes a policy neglecting the delivery delay.
- $Flow.4.1$ and $Flow.4.2$ compute an optimal policy if the demand is non-decreasing and if no disposable is bought two periods after a placement. Algorithm $Flow.4.1$ is a simpler version assuming that there are no close placements.
- $Flow.5.1$ and $Flow.5.2$ compute an optimal policy if at least one container must be idle at each placement. Algorithm $Flow.5.2$ is an extension allowing close placements.

In addition to those, we describe a new algorithm $Opt.D$ computing systematically the optimal solution. We show that taking lead times into account influences the optimal policy tremendously, as it changes the number of containers we will need. Nonetheless, as we explained in Chapter 3, it is debatable whether the use of disposables is meaningful or not, when the demand is deterministic. We include Algorithm $Ndis.D$ into our performance simulations to answer this question.

6.1.1 Optimal Algorithm

We briefly describe our pseudo-polynomial time algorithm $Opt.D$ computing an optimal solution to $DCPP$. We refer to Chapter 7 for further details on the Markov decision process framework we use here.
At each period t, namely each time $(t, 0)$, we denote by $[Y_t, Z_t]$ the state of the system, where Y_t is the number of containers in the manufacturer stock and Z_t is the number of outgoing containers. Since the demand is deterministic and the demand profitability is decreasing at each period, we deduce from Z_t the number of containers used for each demand $D(t-1, r)$ of period $t-1$. Moreover, the state at period $t+1$ follows from the state, the order size, and the purchase size at period t.
We run a backward dynamic program starting from period T where the cost is zero for each state. Then, for each period $t \in [0, T[$ and each state at period t, we compute the optimal ordering and purchase quantities as well as the cost up to the end of the time horizon from the cost of the system at period $t+1$ computed earlier.
When we arrive to period $t = 0$, the cost of the optimal policy is the cost at period 0 starting in state $[0, 0]$.

6.1.2 Simulations

We now compare our algorithms for three different data sets. The first two experiments consider a short time horizon and small increasing demands to compare the performance of the algorithms we developed in Chapters 4 and 5 to the optimal algorithm described in this chapter. The third simulation

Table 6.1: Algorithm performances under five different increasing demand patterns. The running times are given in seconds.

	Pattern:	1	2	3	4	5
Algo *Opt.D*	Cost:	10524	8574	10346	12292	12264
	Duration:	408	180	526	1236	988
Algo *Flow*.4.1	Cost:	10536	8574	10346	**12352**	12264
	Duration:	0.09	0.11	0.09	0.09	0.09
Algo *Flow*.4.2	Cost:	10536	8574	10346	**12352**	12264
	Duration:	0.22	0.28	0.28	0.27	0.28
Algo *Flow*.5.1	Cost:	10524	8574	10346	**12352**	12264
	Duration:	0.11	0.13	0.13	0.11	0.16
Algo *Flow*.5.2	Cost:	10524	8574	10346	**12352**	12264
	Duration:	0.36	0.28	0.33	0.27	0.33
Algo *Ndis.D*	Cost:	13292	11292	14964	18446	16964
	Duration:	0.02	0	0	0	0
Algo *NDel.D*	Cost:	14592	11882	13564	15362	19344
	Duration:	0	0	0	0	0

studies a longer time horizon and general demand patters, and compares the quality of the flow-based policies to each other.
In the first experiment, we consider a short time horizon, with $T = 6$, $R = L_{del} = 3$, and $L_{ord} = 0$. So, there is one early demand and two late demands at each period. For simplicity, every cost is assumed stationary, and we use $C_{man} = 2$, $C_{dis} = 6$, and $C_{cont} = 50$. Table 6.1 presents the experiment results for the following five demand patterns:

1. Pattern 1: $D(t, r) := 15 + 3 \cdot t + r$.
2. Pattern 2: $D(t, r) := 10 + 3 \cdot t + r$.
3. Pattern 3: $D(t, r) := 5 + 2 \cdot (3 \cdot t + r)$.
4. Pattern 4: $D(t, r) := 0 + 3 \cdot (3 \cdot t + r)$.
5. Pattern 5: $D(t, r) := 10 + 3 \cdot (3 \cdot t + r)$.

For instance, in Pattern 3 the initial demand is 5, and the demand increases by 2 at each time step. The costs used for this experiment are $C_{setup} = 1000$ and $C_{dis} = 30$. The flow algorithms always compute the same solution for these test instances, and the generated policies are optimal except for Pattern 4, where the demand starts from zero and increases very fast.
In addition, the policy not using any disposable performs surprisingly poorly, which shows that the disposable option is meaningful even under deterministic demand. We note that ordering a disposable for late demand $D(t, 2)$ incurs a holding cost of 12 at the supplier and of 2 at the manufacturer up

to time $(t+2, 0)$, which total is much lower than the cost 30 of a disposable. However, both the container price and the setup cost are high. Furthermore, even for test instances as small as $T = 6$ and $R = 3$, the optimal algorithm takes several minutes. The two algorithms $Flow.4.1$ and $Flow.5.1$ forbidding close placements have the same running time, and so have Algorithms $Flow.4.2$ and $Flow.5.2$. These four flow-based algorithms compute the same policy.

In the second experiment, we look at the algorithms performance for different setup and disposable costs. The higher the disposable costs, the better Algorithm $Ndis.D$ will perform. Moreover, when the setup costs are low, an optimal policy will purchase new containers very frequently, so The flow algorithms will perform more poorly. Table 6.2 shows simulation results for five different cost pattern. The settings are the same as in the previous experiment, with the fifth demand patterns of Table 6.1. The cost patterns are:

1. Pattern 1: $C_{setup} = 1000$, $C_{dis} = 30$.
2. Pattern 2: $C_{setup} = 1000$, $C_{dis} = 50$.
3. Pattern 3: $C_{setup} = 1000$, $C_{dis} = 100$.
4. Pattern 4: $C_{setup} = 500$, $C_{dis} = 50$.
5. Pattern 5: $C_{setup} = 0$, $C_{dis} = 50$.

For this simulation, the flow algorithms performs also very well, but are not optimal when the setup cost is zero. The algorithms $Flow.4.2$ and $Flow.5.2$ allowing purchasing at consecutive periods perform slightly better, but are still not optimal.

Finally, we compare the flow algorithm for bigger data sets and two kinds of demand patterns, both being non-increasing. We consider the following parameters $T = 20$, $R = 5$ $L_{del} = 5$, $L_{ord} = 0$, and the stationary costs $C_{man} = 2$, $C_{sup} = 6$, $C_{dis} = 75$, $C_{setup} = 10000$, $C_{cont} = 50$.
The first demand pattern describes an approximately increasing demand:

- For each $r \in [0, R[$, demand $D(0, r)$ follows distribution $\mathcal{D}(0, r) := \mathcal{U}(25, 50)$.
- For each $r \in [0, R[, t \in [1, T[$, demand $D(0, r)$ follows the distribution sum $\mathcal{D}(t, r) := \mathcal{D}(t, r-1) + \mathcal{U}(-10, 20)$.

The second demand pattern represents a strictly increasing demand perturbed by several zero values:

- For each $r \in [0, R[$, demand $D(0, r)$ follows distribution $\mathcal{D}(0, r) := \mathcal{U}(25, 50)$.

Table 6.2: Algorithm performances under five different cost patterns. The running times are given in seconds.

	Pattern:	1	2	3	4	5
Algo $Opt.D$	Cost:	12264	15184	16964	14464	13020
	Duration:	1043	1052	1043	1048	1058
Algo $Flow.4.1$	Cost:	12264	15184	16964	14464	**13464**
	Duration:	0.09	0.09	0.09	0.08	0.08
Algo $Flow.4.2$	Cost:	12264	15184	16964	14464	**13200**
	Duration:	0.30	0.30	0.30	0.27	0.27
Algo $Flow.5.1$	Cost:	12264	15184	16964	14464	**13464**
	Duration:	0.11	0.13	0.13	0.13	0.13
Algo $Flow.5.2$	Cost:	12264	15184	16964	14464	**13200**
	Duration:	0.28	0.36	0.30	0.34	0.28
Algo $Ndis.D$	Cost:	16964	16964	16964	15924	13800
	Duration:	0	0	0	0	0
Algo $NDel.D$	Cost:	16344	25120	30936	22620	20120
	Duration:	0	0	0	0	0

Table 6.3: Algorithm performances under general demand patterns. The running times are given in seconds.

	Pattern:	Approx.Increasing	Increasing.with-Zeros
Algo $Flow.4.1$	Cost:	786421	281131
Algo $Flow.4.2$	Cost:	786414	281131
Algo $Flow.5.1$	Cost:	786420	280717
Algo $Flow.5.2$	Cost:	786413	280717
Algo $Ndis.D$	Cost:	796435	288833
Algo $NDel.D$	Cost:	2037442	834853

- For each $r \in [0, R[, t \in [1, T[$, demand $D(0, r)$ follows the distribution sum $\mathcal{D}(t, r) := \mathcal{D}(t, r-1) + \mathcal{U}(5, 10)$.
- In addition, we iteratively and randomly replace 25 demand values with a zero. We have then up to 25 zero demands[2].

Table 6.3 presents the simulation results over 100 instances. Note that in this experiment the policy using zero disposable performs very well, but is still not as good as the flow-based algorithms.
For the first demand pattern, the flow algorithms results are close to each other. Since the cost of Algorithms $Flow.4.1$ and $Flow.4.2$ are not identical, we deduce that the setup cost of 10000 is too low, and therefore it may be

[2]Our algorithm allows to replace a demand with zero which has already been replaced with zero during a previous iteration.

profitable to purchase at consecutive periods. In addition, the algorithms from Chapter 5 perform slightly better than the algorithms from Chapter 4. The reason is that the former algorithms are designed for yielding better results for general demand patterns. Nonetheless, the cost different is not significative and Algorithm *Flow*.4.1 (resp. *Flow*.4.2) performs as well as Algorithm *Flow*.5.1 (resp. *Flow*.5.2) on 98 of the 100 instances.
For the second demand pattern, no optimal policy purchases at consecutive placements, so Algorithms *Flow*.4.1 and *Flow*.4.2 (resp. *Flow*.5.1 and *Flow*.5.2) compute the same policies. Furthermore, Algorithms *Flow*.5.1 and *Flow*.5.2 are significantly faster than their counterpart. Moreover, there is a more noticeable difference between the performances of Algorithms *Flow*.5.1 and *Flow*.5.2. In 33 out of the 100 simulations, Algorithm *Flow*.5.1 performs better than *Flow*.4.1. However, in our experiment Algorithm *Flow*.5.1 performed worse than Algorithm *Flow*.4.1 in three out of 100 test instances.
While there is a clear favor toward Algorithms *Flow*.5.1 and *Flow*.5.2, there are still some text instances where the algorithms from Chapter 4 are more efficient, despite not being thought for this kind of demand pattern. This result highlights the study we made at the end of Chapter 5. This study has shown that Algorithm *Flow*.5.1 actually computes better policies under increasing demand and should in general also perform better under general demand patterns. Nevertheless, there are a few test instances where Algorithm *Flow*.5.1 purchases too many containers and end up performing worse than Algorithm *Flow*.4.1.

Conclusions

In these first experiments, we have analyzed the performance of the algorithms presented in the previous two chapters.
We found that considering disposables noticeably improve the quality of the solution. Moreover, as we expected, a policy neglecting the lead times performs significantly worse than any other policy.
In the previous chapters, we proposed four similar flow-based algorithms: *Flow*.4.1, *Flow*.4.2, *Flow*.5.1, *Flow*.5.2. Algorithms *Flow*.4.2, *Flow*.5.2 are extensions of Algorithms *Flow*.4.1, *Flow*.5.1 that allow close placements, so they perform slightly better than their counterpart. Algorithms *Flow*.4.1 and *Flow*.4.2 were designed in Chapter 4 to handle increasing demand. They perform worse than the other algorithms when the demand has several zero-values. When the demand is not strictly increasing, they may perform better or worse than the other algorithms. Nevertheless, the cost difference is not significant and the algorithms from Chapter 5 perform better in general.
Finally, despite the flow-based algorithms computing very good policies, we are not entirely satisfied with their speed. The rest of this chapter deals with optimizing the execution speed of the flow algorithms.

6.2 Multi-Threading

In this section, we demonstrate that multiple threads can be efficiently used on our algorithms. *Multi-threading* allows to compute different parts of an algorithm simultaneously on different processors or machines. This means significantly decreases the running time, but requires that the algorithm does not need to compute tasks in a specific sequence. Using several threads to solve tasks simultaneously is called *task parallelism*.

In our simulations, the flow algorithm are divided into four parts. Firstly, we generate a data structure representing network $\mathcal{G}$ and which we use to compute flows on networks $\mathcal{H}(k_1, k_2)$ and $\mathcal{F}(k)$. We associate to every node and arc a Boolean value deciding whether it is *active* or not, i.e. whether or not the node or arc belongs to the network $\mathcal{H}(k_1, k_2)$ whose flow we want to compute. We then compute every minimum linear-cost flow on the same network, by considering different active nodes and arcs.

Secondly, for each values k_1 and k_2, we update the nodes and arcs activity to select $\mathcal{H}(k_1, k_2)$ and compute the cost of a minimum linear-cost flow $H^*(k_1, k_2)$.

Thirdly, we use the Wagner-Within dynamic programming framework to deduce the cost of the optimal solution $F^*(T)$ as well as the list of placements. This list can be easily computed by storing for each $F^*(k_2)$ the value k_1 of the best previous placement so that $F^*(k_2) = F^*(k_1, k_2)$.

Finally, we compute the solution by computing and adding the corresponding locally optimal flows $h^*(k_1, k_2)$. We note that we can slightly reduce the running time of the algorithm by saving every value $h^*(k_1, k_2)$, but this would require a much bigger memory space.

Simulations show that the computation of the locally optimal flows $h(k_1, k_2)$ represents 95% of the execution time of Algorithm $Flow$.4.1 (see Table 6.4). The leftover time is used to compute the policy, computing $O(T)$ minimum linear-cost flows. Therefore, any strategy aiming at improving the running time of the algorithm should improve the speed of computing the minimum linear-cost flows.

This experimental result only confirms a simple time complexity analysis. Indeed, we generate network $\mathcal{G}$ in $O(T \cdot R)$ time (see Section 3.4) and the optimal solution cost $F^*(T)$ in $O(T^2)$ time. Meanwhile, the computation of the $H^*(k_1, k_2)$ require $O(T^2)$ minimum linear-cost flow, hence a total time of $O(T^4 \cdot R^2 \log[T \cdot R])$ using the enhanced capacity scaling algorithm. Likewise, generating the optimal solution takes at most $O(T^3 \cdot R^2 \log[T \cdot R])$.

The locally optimal flows $H^*(k_1, k_2)$ can be computed independently, as long as we generate multiple copies of network $\mathcal{G}$. Table 6.5 shows experimental execution speeds of Algorithm $Flow$.4.1 when using several threads to simultaneously compute multiple values $H^*(k_1, k_2)$. We use a machine *Intel(R) Core(TM) i5 CPU* with two cores, take the problem values $T = 25$, $R = 5$ and $L_{del} = 3$, and compute the minimum linear-cost flows with the

Table 6.4: Time consumption of the tasks in Algorithm *Flow*.4.1.

	running time (seconds)	time percentage
Create the network	0.03	0.6%
Compute the $h^*(k_1, k_2)$	45.83	95%
Compute $F^*(T)$	0	0%
Recompute the solution	0.203	4.4%
Total Duration	46.1	100%

enhanced cycle-canceling algorithm. This experiment shows that using a second thread nearly halves the running time. Thus, the program can effectively divide the policy computation between the two cores of the machine. A third thread further decreases the running time, but additional threads have no effect. We conclude:

Conjecture 6.2.1 *If we use a machine with X cores and assign the thread tasks 'Compute $h^*(k_1, k_2)$' with a greedy approach, we can expect the running time to be reduced by factor X.*

Table 6.5: Running time (in seconds) of Algorithm *Flow*.4.1 using 1 to 10 threads to compute the minimum linear-cost flows.

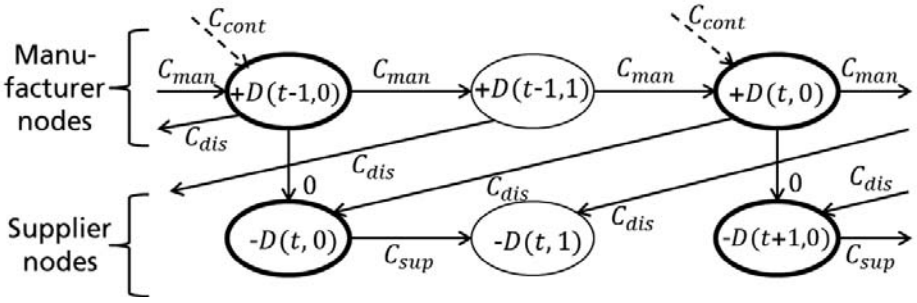
number of threads	1	2	3	4	5	10
running time	46.0	29.9	24.7	23.6	23.7	23.9

6.3 Compact Network Representation

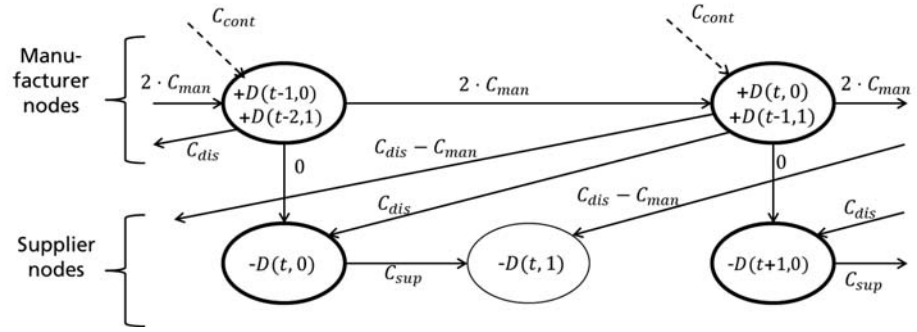
We propose a more compact representation of the network only containing a single manufacturer node per period, at step $r = 0$. We define the *extended network formulation* $\mathcal{G}_e = \mathcal{G}$ as the network described in Chapter 3, and the *compact network formulation* as the new network $\mathcal{G}_c$ we propose in this section. The compact representation can be used for all flow algorithms.

6.3.1 Construction

Once a container arrives to the manufacturer, it must stay there at least until the next ordering decision. Therefore, the number of containers stored in the manufacturer at any point in time (t, r) depends only on the number of arriving full containers and the number of idle containers. Therefore, we can change the network formulation so that each arc $a_{dis}(t, r)$ is incident to a manufacturer node $a_{man}(t_{man}, 0)$ at ordering time, as depicted in Figure 6.1. We recall that we define a network with parameters (V, A, C, E) representing



(a) Extended network pattern.



(b) Compact network pattern.

Figure 6.1: Pattern of the extended and compact network representations.

respectively the set of nodes, the set of arcs, the arc cost function and the node excess function.

Definition 6.3.1 *The compact network* $\mathcal{G}_c := (V_{\mathcal{G},c}, A_{\mathcal{G},c}, C_{\mathcal{G},c}, E_{\mathcal{G},c})$ *is generated from the extended network* $\mathcal{G}_e := (V_{\mathcal{G},e}, A_{\mathcal{G},e}, C_{\mathcal{G},e}, E_{\mathcal{G},e})$ *by aggregating the manufacturer nodes from time* $(t-1,1)$ *to time* $(t,0)$ *for every* $t \in [0,T]$.

1. *For each* $t \in [0,T]$, ***update*** $a_{man}(t,0)$ *to be from* $v_{man}(t)$ *to* $v_{man}(t+1)$ *and with cost:*

$$C_{\mathcal{G},c}\big(a_{man}(t)\big) := \sum_{r} C_{\mathcal{G},e}\big(a_{man}(t,r)\big)$$

2. *For each* $t \in [0,T[$ *and* $r \in [0, R - L_{del}]$, ***update*** *arc* $a_{dis}(t,r)$ *to be from* $v_{man}(t+1,0)$ *to* $v_{sup}(t,r)$ *with cost:*

$$C_{\mathcal{G},c}\big(a_{dis}(t,r)\big) := C_{\mathcal{G},e}\big(a_{dis}(t,r)\big) - \sum_{r_{man}=r+L_{del}}^{R-1} C_{man}(t, r_{man})$$

3. *For each* $t \in [0,T[$ *and* $r \in]R - L_{del}, R[$, ***update*** *arc* $a_{dis}(t,r)$ *to be from* $v_{man}(t+2,0)$ *to* $v_{sup}(t,r)$ *with cost:*

$$C_{\mathcal{G},c}\big(a_{dis}(t,r)\big) := C_{dis}(t,r) - \sum_{r_{man}=r+L_{del}-R}^{R-1} C_{man}(t+1, r_{man})$$

4. *For each* $t \in [1, T+1]$, ***update*** *the excess of* $v_{man}(t,r)$ *into:*

$$E_{\mathcal{G},c}\big(v_{man}(t,0)\big) := E_{\mathcal{G},e}\big(C_{man}(t,0) + \sum_{r=1}^{R-1} C_{\mathcal{G},e}\big(v_{man}(t-1,r)\big)$$

5. *For each* $t \in [0,T]$, ***rename*** $v_{man}(t,0)$ *into* $v_{man}(t)$ *and* $a_{man}(t,0)$ *into* $a_{man}(t)$. ***Rename*** $v_{man}(T+1,0)$ *into* $v_{man}(T+1)$

6. *For each* $t \in [0,T]$ *and* $r \in [1,R[$, ***remove*** *node* $v_{man}(t,r)$ *and arc* $a_{man}(t,r)$ *from the network.*

Remark 6.3.2 *We note that every node (resp. arc) in* $\mathcal{G}_e$ *corresponds to at most one node (resp. arc) in* $\mathcal{G}_c$ *while every node (resp. arc) in* $\mathcal{G}_c$ *corresponds to exactly one node (resp. arc) in* $\mathcal{G}_e$*, due to the operation updating nodes and arcs instead of creating new nodes and arcs (and removing even more of them).*

We now describe algorithm converting a flow on one network representation into a pseudo-flow on the other. We recall that a pseudo-flow is a function of the arc flow whereas a flow is a pseudo-flow so that the imbalance equation holds on every node, i.e. the total outgoing flow equals the total ingoing flow plus the node excess.

Algorithm 13: Transformation of a flow on $\mathcal{G}_e$ into $\mathcal{G}_c$

Data: Flow f_e on $\mathcal{G}_e$
Result: Pseudo-flow f_c on $\mathcal{G}_c$
foreach *arc* $a \in \mathcal{G}_c$ **do**
  $f_c(a) := f_e(a)$;
return f_c;

Lemma 6.3.3 *For each* $t \in [0,T[$ *and all* $r \in [1,R[$*, the flow in arc* $a_{man}(t,r)$ *in any flow* f_e *on* $\mathcal{G}_e$ *is:*

$$f_e\big(a_{man}(t,r)\big) = f_e\big(a_{man}(t,0)\big) + \sum_{r'=1}^{r} \Big[E_{\mathcal{G},e}(v_{man}(t,r')) - f_e\big(a_{dis}(t,r'-L_{del})\big)\Big] \tag{6.1}$$

Proof:

For each $t \in [0,T[$ and for $r = 1$, the equation follows from the imbalance property. We deduce the lemma for each $t \in [0,T[$ by recurrence on r.

□

Aggregating the manufacturer nodes induces a flow of $E_{\mathcal{G},e}\big(v_{man}(t,r)\big)$ from node $v_{man}(t,r)$ to node $v_{man}(t+1,0)$, for each $t \in [0,T]$ and each $r \in [1,R[$.

We denote this cost by $C^0_{\mathcal{G},c}$:

$$C^0_{\mathcal{G},c} := \sum_{t=0}^{T} \Big(\sum_{r=1}^{R} E_{\mathcal{G},e}\big(v_{man}(t,r)\big) \cdot \sum_{r'=r}^{R-1} C_{man}(t,r') \Big) \tag{6.2}$$

Lemma 6.3.4 *Algorithm 13 transforms a flow f_e on $\mathcal{G}_e$ into a flow f_c on $\mathcal{G}_c$ so that the following relationship holds between their costs F_c and F_e:*

$$F_e = F_c + C^0_{\mathcal{G},c} \tag{6.3}$$

Proof:

Consider a flow f_e on $\mathcal{G}_e$, and the corresponding pseudo-flow f_c generated by Algorithm 13. Since f_c is generated from f_e by aggregating some nodes, this pseudo-flow respects the imbalance property at every node in $\mathcal{G}_c$ and is hence an actual flow on f_c.
We now consider the difference of cost between these two flows. For each arc a, we denote by $F_e(a)$ the cost of the flow $f_e(a)$, by $F_c(a)$ the cost of the flow $f_c(a)$ and by $\Delta_{c,e}(a)$ the difference in cost:

$$\Delta_{c,e}(a) := F_e(a) - F_c(a)$$

where:

$$\forall a \notin \mathcal{G}_c, F_c(a) = 0$$

The difference of cost between the arcs $a_{dis}(t,r)$ is then:

$$\forall t \in [0,T[,\ \forall r \in [0, R - L_{del}] :$$

$$\Delta_{c,e}\big(a_{dis}(t,r)\big) := f_e\big(a_{dis}(t,r)\big) \cdot \sum_{r'=r+L_{del}}^{R-1} C_{man}(t,r')$$

$$\forall t \in [0,T[,\ \forall r \in [0, R - L_{del}] :$$

$$\Delta_{c,e}\big(a_{dis}(t,r)\big) := f_e\big(a_{dis}(t,r)\big) \cdot \sum_{r'=r+L_{del}}^{R-1} C_{man}(t+1,r')$$

The difference of cost between the arcs $a_{man}(t,0)$ and $a_{man}(t)$ is:

$$\forall t \in [0,T[:\ \Delta_{c,e}\big(a_{man}(t)\big) = -f_e\big(a_{man}(t,0)\big) \cdot \sum_{r=1}^{R-1} C_{man}(t,r)$$

For each $t \in [0,T[$ and all $r \in [1,R[$, the flow in arc $a_{man}(t,r)$ can be computed using the imbalance property and the flow before time (t,r):

$$f_e\big(a_{man}(t,r)\big) = f_e\big(a_{man}(t,0)\big) + \sum_{r'=1}^{r} \Big[E_{\mathcal{G},e}(v_{man}(t,r')) - f_e\big(a_{dis}(t, r' - L_{del})\big) \Big]$$

The total cost difference between these two flows is then:

$$\begin{aligned}
\Delta_{c,e} := \sum_{t=0}^{T-1} \Big(& \sum_{r=0}^{R-L_{del}} \big[f_e\big(a_{dis}(t,r)\big) \cdot \sum_{r'=r+L_{del}}^{R-1} C_{man}(t,r') \big] \\
& + \sum_{r=R-L_{del}+1}^{R-1} \big[f_e\big(a_{dis}(t,r)\big) \cdot \sum_{r'=r+L_{del}}^{R-1} C_{man}(t+1,r') \big] \\
& - f_e\big(a_{man}(t,0)\big) \cdot \sum_{r=1}^{R-1} C_{man}(t,r) \\
& + \sum_{r=1}^{R-1} \Big[\sum_{r'=1}^{r} \big[E_{\mathcal{G},e}(v_{man}(t,r')) - f_e\big(a_{dis}(t,r'-L_{del})\big) \big] \\
& \qquad + f_e\big(a_{man}(t,0)\big) \Big] \cdot C_{man}(t,r) \Big) \\
= \sum_{t=0}^{T-1} \Big(& \sum_{r=0}^{R-1} \sum_{r'=1}^{r} E_{\mathcal{G},e}\big(v_{man}(t,r')\big) \cdot C_{man}(t,r) \Big) \\
= \sum_{t=0}^{T} \Big(& \sum_{r=1}^{R} E_{\mathcal{G},e}\big(v_{man}(t,r)\big) \cdot \sum_{r'=r}^{R-1} C_{man}(t,r') \Big) = C^0_{\mathcal{G},c}
\end{aligned}$$

□

Algorithm 14: Transformation of a flow on $\mathcal{G}_c$ into $\mathcal{G}_e$

```
// Set the flow to common arcs
foreach arc a ∈ G_e ∩ G_c do
    f_e(a) := f_c(a);
// Set an empty flow to new arcs
foreach arc a ∈ G_e − G_c do
    f_e(a) := 0;
// Update the flow in new arcs
for t : 0 → T do
    for r : 1 → R − 1 do
        f_e(a_man(t, r)) := leftover imbalance on v_man(t, r);
return f_e;
```

Lemma 6.3.5 *Algorithm 14 transforms a flow f_c on $\mathcal{G}_c$ into a flow f_e on $\mathcal{G}_e$ so that the following relationship holds between their costs F_c and F_e:*

$$F_c = F_e + C^0_{\mathcal{G},c} \tag{6.4}$$

Proof:

Consider a flow f_c on $\mathcal{G}_c$ and use Algorithm 14 to transform it into a pseudo-flow f_e on $\mathcal{G}_e$. By Lemma 6.3.4, f_e respects the imbalance property

on every node besides maybe the nodes $v_{man}(t,0)$. The imbalance property of these nodes $v_{man}(t,0)$ follows from the fact that f_c is a flow and no two nodes $v_{man}(t_1,0)$ and $v_{man}(t_2,0)$ are adjacent. Furthermore, Algorithm 13 transforms f_e back into f_c. We deduce that Equation (6.4) holds.

□

We deduce that both network representations are equivalent, so we can use the compact network to find an optimal solution to $DCPP$.

Proposition 6.3.6 *Both network flow formulations $\mathcal{G}_e$ and $\mathcal{G}_c$ are equivalent: Algorithm 13 transforms any flow f_e on $\mathcal{G}_e$ in to a flow f_c on $\mathcal{G}_c$ and Algorithm 13 makes the reverse operation. Their costs F_e and F_c are so that:*

$$F_c = F_e + C^0_{\mathcal{G},c} \tag{6.5}$$

6.3.2 Predicted Improvement

We recall that the extended network contains $N_e(T,R) := R \cdot (2 \cdot T + 3) + 2$ nodes and $M_e(T,R) := (T+1) \cdot (3 \cdot R + 1) + 1$ arcs.

Proposition 6.3.7 *The compact network has $N_c(T,R) := N_e - (T+1) \cdot (R-1)$ nodes and $M_c(T,R) := M_e - (T+1) \cdot (R-1)$ arcs. Using the compact representation, Algorithms Flow.4.1 and Flow.4.2 are expected to be around $\pi := (M_c \cdot N_c)/(M_e \cdot N_e) \in [1,6]$ times faster.*

Proof:

At each period $t \in [0,T]$, we remove $R-1$ nodes $v_{man}(t,r)$ and $R-1$ arcs $a_{man}(t,r)$, with $r \in [1,R[$. Moreover, we saw earlier in this chapter that the most time consuming task in our network resolution is the minimum linear-cost flow. The enhanced capacity scaling algorithm takes $O(M \cdot N \cdot \log[N]^2) = o(M \cdot N)$ time. We can thus expect the running time of the algorithm to be $(M_e \cdot N_e)/(M_c \cdot N_c)$ times faster. Finally, we have $1 \leq N_e/N_c \leq 2$ and $1 \leq M_e/M_c \leq 3$, so $\forall T, R,\ \pi(T,R) \in [1,6]$

□

Figure 6.2 illustrates the function $\pi(T,R)$ for different values of T and R. When $R = 1$, $\pi(T,R) = 1$ so our algorithms have a similar running time for both network representations. For $(T,R) = (100,20)$, $\pi(T,R) \approx 3,5$, so the compact networks are expected to take only 30% of the time required to compute the solution with the extended network.

Example 6.1 *Consider a container purchasing problem over one year and with weekly container ordering. There are 53 weeks and up to 7 working days a week. Thus the compact representation only keeps 58% of the nodes and 73% of the arcs, going from $N_e = 765$ nodes and $M_e = 1189$ arcs in $\mathcal{G}_e$ to $N_c = 441$ nodes and $M_c = 865$ arcs in the compact network. We hence*

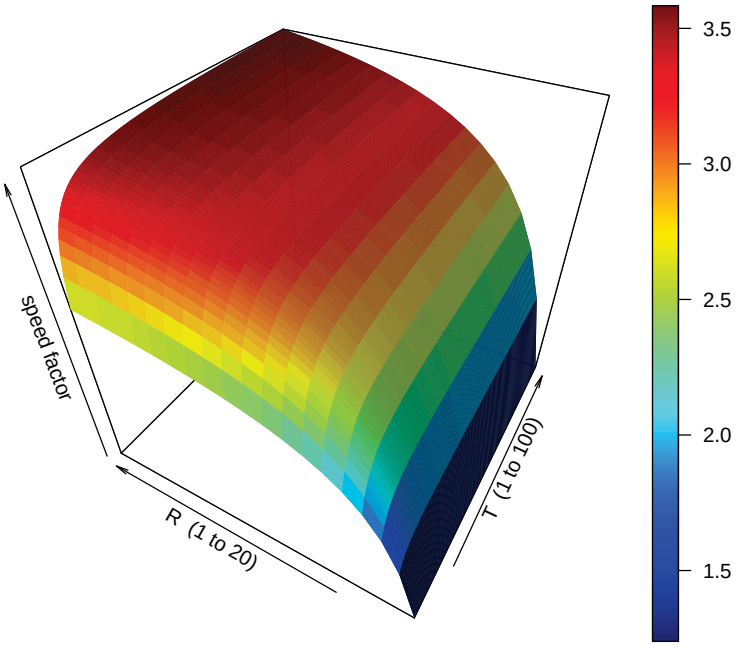


Figure 6.2: Evolution of the expected speed factor $\pi(T, R)$ for $T \in [1, 100]$ and $R \in [1, 20]$.

go from $M_e \cdot N_e \cdot \log[N_e]^2 = 40.101.739$ *to* $M_c \cdot N_c \cdot \log[N_c]^2 = 14.143.375$, *which is only roughly* 35% *of the time.*

Conjecture 6.3.8 *In practice, the flow algorithms are expected to be two or three times faster when using the compact network formulation instead of the extended one.*

6.4 Cycle-Canceling Algorithm

Our flow algorithms solve the *DCPP* by computing several minimum linear-cost flows. As mentioned in Chapter 3, the fastest minimum linear-cost flow algorithm in the literature is the enhanced capacity scaling algorithm from Orlin [85]. We have seen in Section 6.2 that the running time of our flow algorithms directly depends on the performance of the minimum linear-cost flow resolutions. In this section, we develop an alternative algorithm specific to our network structure. Firstly, we briefly present how we could improve the capacity scaling algorithms. Then, we present our cycle-canceling framework for Algorithm *Flow*.4.1. Afterward, we analyze the time complexity

and propose an efficient way of removing the desired cycle in linear time. Finally, we shortly explain how to extend the cycle-canceling framework to the other three flow algorithms

6.4.1 Remark on the Capacity Scaling Algorithms

We briefly describe two possible improvements of the capacity scaling algorithms, but do not test them, as we present a more efficient algorithm.
Firstly, in contrast to a random network, in our flow networks $\mathcal{H}(k_1, k_2)$ most sinks are nodes $v_{sup}(t, r)$ and are adjacent to a source $v_{man}(t, r + L_{del})$ with the exact opposite excess. Thus, instead of using a capacity scaling algorithm, we can send a flow of $D(t, r)$ from $v_{man}(t, r + L_{del})$ to $v_{sup}(t, r)$, for all node $v_{sup}(t, r)$ in the considered network. For Algorithms $Flow.5.1$ and $Flow.5.2$, the only nodes left with positive imbalance are v_{pur}, $v_{man}(k_2, 0)$ and $v_{man}(k_2 + 1, 0)$, due to the operation reverting the arcs with negative cost. For Algorithms $Flow.4.1$ and $Flow.4.2$, the nodes left are the manufacturer nodes corresponding to a demand at period k_1 and $k_1 + 1$, the supplier nodes at periods k_2 and $k_2 + 1$, plus possibly v_{pur}. In both cases, we can easily set the major part of the optimal flow.
Secondly, we can improve the Dijkstra algorithm computing the shortest path from a source to every other node. The Dijkstra algorithm assumes that each arc has positive cost, and divides the set of nodes into three groups:

1. The nodes whose distances are computed; we call these nodes *achieved*.

2. The nodes in which we already found a path from the source; we call these nodes *only-visited*.

3. The other node, which we call *unvisited*.

Consider the only-visited node with the minimum-length computed path. We can easily check that this path is the shortest path. The Dijkstra algorithm marks it as achieved, and expends the computed paths and the only-visited nodes using its incident arcs. The Dijkstra algorithm must hence repetitively find the best only-visited node $O(n)$ times, which takes a lot of time if there are many only-visited nodes at the same time. Our network structure is so that the nodes from period t are only connected to nodes of periods $t - 1$ and $t + 1$. Thus, if we visited some nodes at periods $t - 2$ and $t + 2$ for some period t, it is very likely that the unvisited nodes at period t are isolated from the other unvisited nodes at periods before $t - 1$ and after $t + 2$, so we should be able to compute their distances separately. Consequently, we should be able to reduce the experimental running time of the algorithm by $O(\log[T]/\log[R])$, keeping only $O(R)$ only-visited nodes.

6.4.2 Adapted Cycle-Canceling

We consider Algorithm *Flow*.4.1 from Chapter 4, and two consecutive placements k_1 and k_2. We assume in particular that Hypotheses 2.1 [Delay], 2.2 [Cost], 4.1 [Demand], 4.2 [Placement] and 4.4 [Distance] hold. We use the extended network formulation proposed in Chapter 3. Our objective is to compute the minimum linear-cost flows $h^*(k_1, k_2)$.
Without loss of generality, we assume that the purchasing cost $C_{cont}(k_2)$ at period k_2 is lower than the purchasing cost $C_{cont}(k_1)$ at period k_1 plus the total manufacturer holding cost of an idle container from period k_1 to period $k_2 - 1$. Otherwise, it would not be optimal to purchase containers at period k_2 when we purchase container at period k_1; we could replace $h^*(k_1, k_2)$ with any sub-optimal flow on $\mathcal{H}(k_1, k_2)$, and the algorithm would still generate an optimal policy.
Similarly to the minimum mean cycle-canceling algorithm, we look for the best negative cycle to remove, so that we do not need to cancel too many negative cycles. Our approach is based on the following lemma:

Lemma 6.4.1 *Consider network $\mathcal{H}(k_1, k_2)$ for some $0 \leq k_1 < k_2 \leq T$. If $k_2 < T$, the container fleet size at period $k_2 + 1$ is:*

$$u_{k_2+1} := \sum_{r=1-L_{del}}^{R-1} D(k_2 + 1, r) \tag{6.6}$$

Moreover, if every container is purchased at period k_1, then the system cost is minimized when no disposable is bought. If $k_2 = T$ and u_{T-1} or more containers are purchased at period k_1, then the system cost is minimized when no disposable is bought.

Proof:
Suppose that $k_2 < T$. At period t, we have up to $\sum_{r=1-L_{del}}^{R-1} D(t-1, r)$ outgoing containers and order up to $\sum_{r=0}^{R-1} D(t, r)$. Therefore, we need at most $\sum_{r=1-L_{del}}^{R-1} D(t, r)$ containers in the system at period t. By Hypothesis 4.1 [Demand], we do not need more than u_{k_2+1} containers. By Hypothesis 4.2 [Placement], no disposable is bought at periods k_2 and k_2+1, we need at least u_{k_2+1} containers. We conclude that after the last purchasing at period k_2, we will have exactly u_{k_2+1} containers.
Moreover, by demand profitability no disposable is bought if some containers are idle at the same time. By Hypothesis 4.1 [Demand], if every container is purchased at period k_1, we have enough containers to fulfill every demand, so we do not need any disposable.
The reasoning is the same for $k_2 = T$.

□

We hence define a *best flow* as a flow with an optimal ordering policy given a purchasing policy. Our algorithm starts with the best flow purchasing every

containers at period k_1, and then iteratively shift the container purchasing to period k_2 while conserving an optimal ordering policy. For $i \geq 0$, we define $h^i(k_1, k_2)$ the flow on $\mathcal{H}(k_1, k_2)$ at the beginning of the i-th iteration, before looking for a negative cycle. We generate the initial flow $h^0(k_1, k_2)$ using Algorithm 15, and define invariant (I) to hold at the beginning of each iteration so that the flow is a best flow.

Invariant (I): In the flow $h^i(k_1, k_2)$ at iteration i, any negative cycle contains arc $a_{pur}(k_1)$ (hence node v_{pur}).

Algorithm 15: Generating the Initial Flow $h^0(k_1, k_2)$

for *Supplier Node* $v_{sup}(t, r)$ **do**
  Send $D(t, r)$ flow units from v_{pur} to $v_{sup}(t, r)$ via $a_{pur}(k_1)$, $a_{man}(t', r')$ for all $(k_1, 0) \leq (t', r') < (t, 0)$, $a_{ord}(t)$ and $a_{sup}(t, r')$ for all $0 \leq r' < r$;

foreach *Manufacturer node* $v_{man}(t, r)$ **do**
  Send all excess from $v_{man}(t, r)$ to v_{pur} via $a_{man}(t', r')$ for $(t, r) \leq (t', r') < (T, R-1)$ and $a_{end}(k_2 + 1)$;

return the generated flow;

Lemma 6.4.2 *The pseudo-flow generated by Algorithm 15 is a best flow.*

Proof:

The pseudo-flow is generated by having the imbalance property respected of every manufacturer nodes and every supplier nodes while only increasing the flow in the arcs. Since the network imbalance is zero, the imbalance of v_{pur} is also zero, so we have generated a flow. By Lemma 6.4.1, it is thus a best flow.

□

We now show how to remove negative cycles while keeping the invariant.

Remark 6.4.3 *Every cycle in* $\mathcal{H}(k_1, k_2)$ *contains at least either a disposable arc, arc* $a_{pur}(k_1)$ *or* $a_{pur}(k_2)$*, since the network is acyclic without these arcs.*

We recall that a negative cycle has a direction such that adding a flow unit in this direction of the cycle decreases the flow cost. An arc in a cycle is called *forward* if it is in the direction of the cycle and *backward* otherwise.

Lemma 6.4.4 *Flow* $h^0(k_1, k_2)$ *respects invariant* (I).

Proof:

Suppose that there is a negative cycle in $h^0(k_1, k_2)$ not containing $a_{pur}(k_1)$. Since the flow in $a_{pur}(k_1)$ is empty, the cycle does not contain v_{pur}. By Remark 6.4.3, there is at least one disposable arc in the cycle. We consider

the negative cycle with the shortest length. In particular, this negative cycle is not the union of smaller cycles. If this cycle contains a single disposable arc $a_{dis}(t, r)$, then the situation is as illustrated in Figure 6.3.

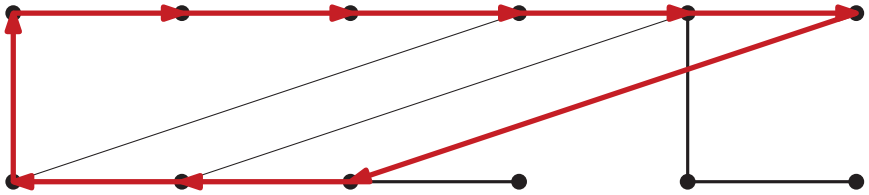


Figure 6.3: A negative cycle (in red) with a single disposable arc and without arc $a_{pur}(k_1)$ contradicts the assumption of demand profitability.

Let $v_{man}(t_{man}, r_{man})$ be the manufacturer node incident to $a_{dis}(t, r)$. Necessarily, the cycle contains arc $a_{ord}(t)$ and arcs $a_{sup}(t, r')$ for each $r \in [0, r[$. The only way to get a cycle is to also have arcs $a_{man}(t', r')$ for each $(t, 0) \leq (t', r') < (t_{man}, r_{man})$. Since $h^0(k_1, k_2)$ does not contain $a_{dis}(t, r)$, this disposable arc is necessarily forward and it directly follows that keeping a container idle instead of fulfilling demand $D(t, r)$ decreases the cost. Thus the demand is not profitable, which contradicts our hypothesis.
We deduce that there are at least two disposable arcs. These arcs must all be forward arcs in the negative cycle. In particular, because the length of the negative cycle is minimum, this implies that the negative cycle cannot contain disposable arcs relative to the demands of the same period.
Let (t_1, r_1) and (t_2, r_2) be the two earliest times so that arc $a_{dis}(t_1, r_1)$ and $a_{dis}(t_2, r_2)$ are in the cycle, and assume that $t_1 < t_2$. Necessarily, the negative cycle contains arc $a_{ord}(t_1)$, $a_{ord}(t_2)$ as well as arcs $a_{sup}(t_1, r')$ for $r' \in [0, r_1[$ and $a_{sup}(t_2, r')$ for $r' \in [0, r_2[$. If we had $t_2 = t_1 + 1$, then the negative cycle would be as shown in Figure 6.4, so $a_{dis}(t_1, r_1)$ and $a_{dis}(t_2, r_2)$ are in opposite direction, which is a contradiction.

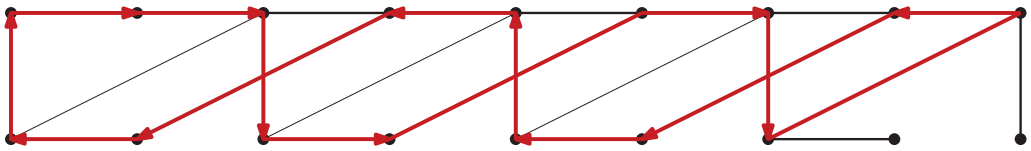


Figure 6.4: A negative cycle (in red) with several disposables arc and without arc $a_{pur}(k_1)$ necessarily contains backward disposable arcs.

Thus, we have $t_2 > t_1 + 1$ and $a_{ord}(t_1 + 1)$ is not in the negative cycle. So, the cycle contains $a_{man}(t_1 + 1, r')$ for $(t + 1, 0) \leq (t + 1, r') \leq (t, r_1) + L_{del}$. Therefore, the negative cycle contains a smaller cycle with a single disposable arc $a_{dis}(t_1, r_1)$, hence does not have minimum length.

□

Lemma 6.4.4 states that invariant (I) holds at the first iteration. We now want to remove negative cycles while preserving (I). Our approach is to

compute a cycle containing arc $a_{pur}(k_1)$ backward and with *minimum cost-per-flow-unit*, i.e. with minimum cost assuming that we only send one flow unit through the cycle. If the resulting cycle has negative cost, then we remove it and call $h^{i+1}(k_1, k_2)$ the new flow. Otherwise, we stop our algorithm and show in the following that $h^i(k_1, k_2) = h^*(k_1, k_2)$ is locally optimal. We denote by $\mathcal{R}\big(h(k_1, k_2)\big)$ the residual network of flow $h(k_1, k_2)$ on $\mathcal{H}(k_1, k_2)$.

Remark 6.4.5 *Consider the flow $h^i(k_1, k_2)$ and suppose that $a_{pur}(k_1)$ has a positive flow. A minimum cost-per-flow-unit cycle including $a_{pur}(k_1)$ backward is by definition a shortest path from node v_{pur} to node $v_{man}(k_1, 0)$ in the residual network $\mathcal{R}(h^i(k_1, k_2))$.*

Remark 6.4.6 *There may be negative cycles with a higher cost-per-flow-unit reducing the solution cost by a greater amount due to a greater flow capacity in the cycle.*

Remark 6.4.7 *If a flow contains a negative cycle with $a_{pur}(k_1)$ backward, then this cycle has a negative cost per-flow-unit, so the minimum cost-per-flow-unit cycle containing $a_{pur}(k_1)$ backward is negative.*

Lemma 6.4.8 *Removing a minimum cost-per-flow-unit negative cycle containing $a_{pur}(k_1)$ backward preserves the invariant.*

Proof:

Suppose that the invariant holds for $h^i(k_1, k_2)$ but not for $h^{i+1}(k_1, k_2)$. We denote by $nc_i(k_1, k_2)$ the negative cycle flow so that $h^{i+1}(k_1, k_2) = h^i(k_1, k_2) + nc_i(k_1, k_2)$. Since the invariant does not hold at iteration $i + 1$, there is a negative cycle nc_0 in $h^i(k_1, k_2)$.
Suppose first that the negative cycle excludes v_{pur}, i.e. does not contain $a_{pur}(k_2)$ and $a_{end}(k_2{+}1)$. Since there is no negative cycle in $h^i(k_1, k_2)$ excluding v_{pur}, some arcs in the negative cycle must be contained in $nc_{i-1}(k_1, k_2)$. Thus, we can include the negative cycle nc_0 into $nc_{i-1}(k_1, k_2)$. The resulting path has a smaller cost-per-flow-unit than $nc_{i-1}(k_1, k_2)$, so $nc_{i-1}(k_1, k_2)$ cannot be not a shortest path.
Suppose now that the negative cycles includes $a_{pur}(k_2)$ and $a_{end}(k_2+1)$, but not $a_{pur}(k_1)$. Recall that no disposable is allowed at period k_2, so the only path from $v_{man}(k_2)$ to $v_{man}(k_2 + 1)$ necessarily goes through $a_{man}(k_2, r)$ for $r \in [0, R[$, so there is a unique cycle. This cycle has negative cost when $a_{pur}(k_2)$, $a_{end}(k_2 + 1)$ and the $a_{man}(k_2, r)$ are all backward and contains a positive flow. Since the starting flow in $a_{pur}(k_2)$ is empty, we must have previously removed a negative cycle containing $a_{pur}(k_2)$ forward and $a_{pur}(k_2)$ backward. This cycle cannot contain $a_{end}(k_2 + 1)$ as v_{pur} is only incident to three arcs while we only remove negative cycles going at most once in each arc. Thus, this cycle does not go in arcs $a_{man}(k_2, r)$ for $r \in [0, R[$

either. This cycle is does not minimize the cost-per-flow-unit, because it contains arc $a_{pur}(k_2)$ forward with positive cost, while arcs $a_{end}(k_2+1)$ and the $a_{man}(k_2, r)$ backward induce a negative flow, and we can use them as they have a positive flow. We conclude that this previous cycle was not the best one.

□

Our *Adapted Cycle-Canceling Algorithm* is as following:

Algorithm 16: Adapted Cycle-Canceling Framework

Data: values k_1, k_2
Result: minimum linear-cost flow on $\mathcal{H}(k_1, k_2)$
$h := h^0(k_1, k_2)$;
$nc :=$ minimum cost-per-flow-unit cycle in h on $\mathcal{H}(k_1, k_2)$;
$\delta :=$ maximum flow in nc;
while *nc has negative cost* **do**
  $h := h + \delta \cdot nc$: remove the cycle nc from flow h;
  $nc :=$ minimum cost-per-flow-unit cycle in h on $\mathcal{H}(k_1, k_2)$;
  $\delta :=$ maximum flow in nc;
$h^*(k_1, k_2) := h$;

Corollary 6.4.9 *Invariant (I) holds at the beginning of each iteration of Algorithm 16.*

Corollary 6.4.10 *If there is no negative cycle containing $a_{pur}(k_1)$ backward, then there is no better solution purchasing less containers at period k_1.*

To prove the correctness of our adapted cycle-canceling algorithm, we have yet to show that the algorithm stops with the right number of containers purchased at period k_1.

Lemma 6.4.11 *No flow on $\mathcal{H}(k_1, k_2)$ has a lower cost than $h^0(k_1, k_2)$ while purchasing more containers at period k_1.*

Proof:

Consider the minimum cost flow on $\mathcal{H}(k_1, k_2)$ among the flows purchasing more containers at period k_1. By Lemma 6.4.1, no disposables is bought, so the only different to flow $h^0(k_1, k_2)$ is that more containers are purchased and idle on the whole time interval $[k_1, k_2 + 1]$, so the cost is greater.

□

Proposition 6.4.12 *Suppose that invariant (I) holds at iteration $i \geq 0$, there is a negative cycle containing $a_{pur}(k_1)$ backward and there is no flow with lower cost while purchasing more containers at period k_1 than $h^i(k_1, k_2)$.*

Then at iteration $i+1$, there is no flow with lower cost than $h^{i+1}(k_1, k_2)$ while purchasing more containers at period k_1.

Proof:

We consider the flow $h^i(k_1, k_2)$ and the negative cycle $nc_i(k_1, k_2)$ we computed. Let $nflow_i$ be the quantity of flow we removed to transform $h^i(k_1, k_2)$ into $h^{i+1}(k_1, k_2)$. Then, for every value $n \in [0, flow_i - 1]$, The flow generated from $h^i(k_1, k_2)$ by removing n flow units in cycle $nc_i(k_1, k_2)$ has exactly the same arcs with positive flow, hence exactly the same negative cycle. Consequently, invariant (I) also holds for this flow and there is no flow with the same purchase size at period k_1 with lower cost.
Moreover, this flow still contains the negative cycle $nc_{i-2}(k_1, k_2)$, so it has a greater cost than the new flow $h^{i+1}(k_1, k_2)$.

□

Theorem 6.4.13 *At iteration i, if there is no negative cycle including arc $a_{pur}(k_1)$, then the flow $h^i(k_1, k_2)$ is locally optimal.*

Proof:

This result follows from Lemma 6.4.11, Proposition 6.4.12 and Corollary 6.4.9.

□

6.4.3 Time Complexity Analysis

Lemma 6.4.14 *Flow $h^0(k_1, k_2)$ can be computed in $O(R \cdot T)$ time.*

Proof:

The construction of $h^0(k_1, k_2)$ is divided into three steps:

1. Send the manufacturer excess to the supplier nodes,
2. Fulfill the imbalance of supplier nodes before period k_2 via arc $a_{pur}(k_1)$.
3. Fulfill either the imbalance of manufacturers starting from period $k_2 = T$ via arc $a_{end}(k_2 + 1)$, or the imbalance of suppliers from periods $k_2 < T$ and $k_2 + 1$ via arc $a_{pur}(k_2)$.

In the first step, we send a flow over a predefined path of length $O(R)$ for each supplier node, which should take in total $O(R^2 \cdot T)$ time. However, we can improve this complexity to $O(R \cdot T)$ by grouping the flow of the supplier nodes $v_{sup}(t, r)$ of a same period t:

- We first send all the excess of the manufacturer nodes $v_{man}(t', r')$ with $(t-1, 1) \leq (t', r') \leq (t, 0)$ to supplier node $v_{sup}(t, 0)$ in $O(R)$ time.

- We do it by sending iteratively all the excess from node $v_{man}(t', r')$ to node $v_{man}(t', r' + 1)$ in $O(1)$ time and by increasing $r' \in [1, R[$, then all the excess in $v_{man}(t, 0)$ to $v_{sup}(t, 0)$.
- We get for each r in $O(1)$ time the flow quantity to send to $v_{sup}(t, r)$.
- We thus send in $O(1)$ time the desired flow from $v_{sup}(t, r)$ to $v_{sup}(t, r+1)$ by increasing $r \in [0, R-2]$. The total time is then $O(R)$

The second step consists in sending a flow over paths of length $O(R \cdot T)$ and to $O(R \cdot T)$ supplier nodes. This would lead to a complexity of $O(R^2 \cdot T^2)$. Similarly to the first step, we improve this complexity by pre-computing for each period t the quantity of flow to send over arc $a_{ord}(t)$ in $O(R)$ time. Then we deduce the quantity of flow that must go through every arc $a_{man}(t, r)$ in $O(T \cdot R)$ time. Finally, we can send in $O(R, T)$ time the flow from $v_{sup}(t, 0)$ to the $v_{sup}(t, r)$ for each t and r. The complexity is then $O(R \cdot T)$ time.
The last step is similar to the second one, but takes $O(R)$ time.

□

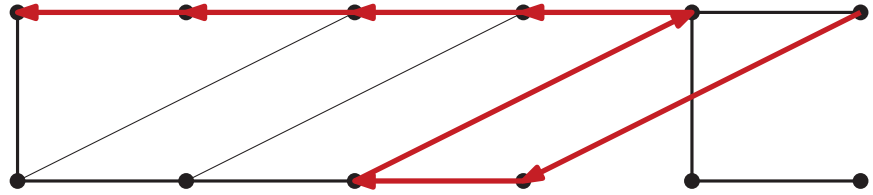
Lemma 6.4.15 *Under invariant (I), every disposable arc a_{dis} in a minimum cost-per-flow-unit cycle containing $a_{pur}(k_1)$ backward is forward.*

Proof:

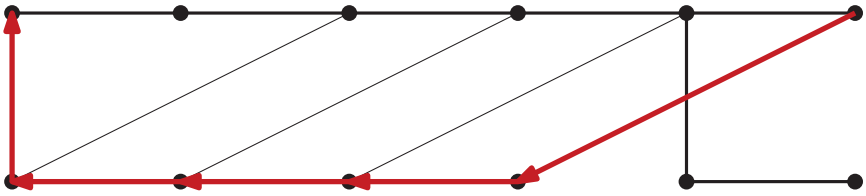
This proof is similar to the proof of Lemma 6.4.4. The lemma trivially holds for the first iteration. Suppose that a minimum cost-per-flow-unit cycle including $a_{pur}(k_1)$ backward contains a backward arc. Under invariant (I), the cycle cannot contain any smaller negative cycle, as it would exclude v_{pur}. Therefore, without loss of generality, we assume that the cycle is simple, i.e. does not contain any smaller cycle.
Suppose first that the cycle contains two disposable arcs $a_{dis}(t, r_1)$ and $a_{dis}(t, r_2)$ at the same period, as illustrated in Figure 6.5. The minimum cost-per-flow-unit cycle contains a path as in Figure 6.5, which is the sum of another path and a cycle. This cycle has thus necessarily a negative cost[3], which contradicts the demand profitability assumption. Thus, the minimum cost-per-flow-unit cycle cannot contain two disposable arcs at the same period.
Let $a_{dis}(t_{sup}, r_{sup})$ denote the earliest backward disposable arc. This last analysis is illustrated in Figure 6.6. Then, the cycle contains $a_{ord}(t_{sup})$ forward and $a_{sup}(t_{sup}, r)$ forward for $r < r_{sup}$, which we will call the 'red path'. Let $v_{man}(t_{man}, r_{man})$ be the manufacturer node incident to $a_{dis}(t_{sup}, r_{sup})$. In addition, the minimum cost-per-flow-unit cycle including $a_{pur}(k_1)$ backward necessarily contains two other paths:

1. a 'blue path' from v_{pur} to $v_{man}(t_{sup}, 0)$, and

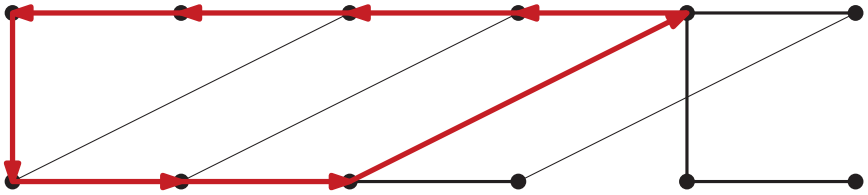
[3]because it is included in a minimum cost cycle



(a) Path containing two disposable arcs in the same period.



(b) Decomposition (1/2): simple path



(c) Decomposition (2/2): cycle with positive cost

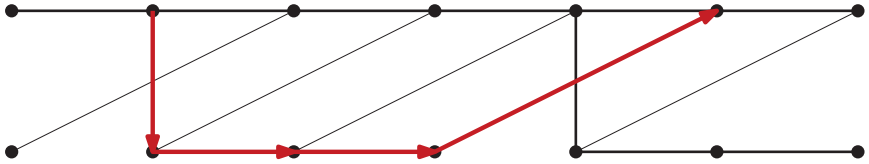
Figure 6.5: A minimum cost-per-flow-unit cycle containing disposable arcs in the same period violate the demand profitability assumption. The path (in red) shown in Figure 6.5a can be decomposed into a path (cf. Figure 6.5b) and a positive cycle (cf. Figure 6.5c).

2. a 'brown path' from $v_{man}(t_{man}, r_{man})$ to $v_{man}(k_1, 0)$.

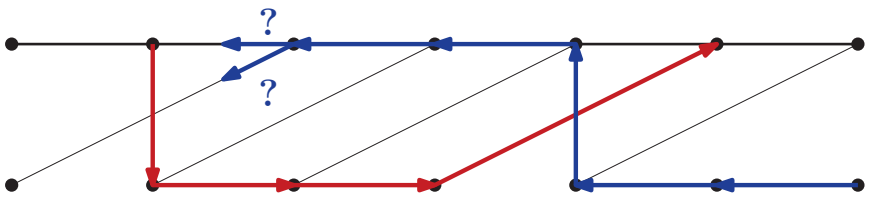
By assumption, the minimum cost-per-flow-unit cycle is simple, so these three paths are disjoint. For the blue and the red paths to be disjoints, we need arc $a_{dis}(t_{sup}, r_{sup})$ corresponds to a late demand and the blue path to contain arc $a_{ord}(t_{sup} + 1)$ backward, as well as arcs $a_{man}(t_{sup}, r)$ backward for $r \in [L_{del}, R[$. However, for the brawn path disjoint to the red path, it need to go through a disposable arc $a_{dis}(t_{sup} - 1, r)$ for some $r \leq L_{del}$, which requires to visit arc $a_{man}(t_{sup}, L_{del})$, which already belongs to the blue path. This is a contradiction, and we conclude that no disposable arc can be backward in the cycle.

□

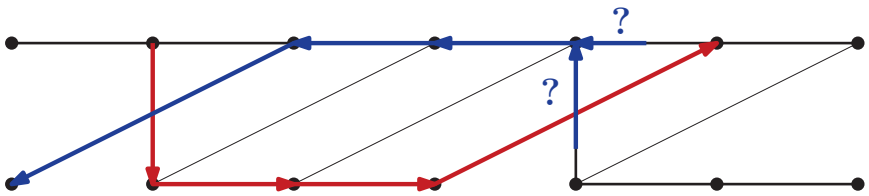
Lemma 6.4.15 states that, when we decrease the number of containers purchased at placement k_1, we never decrease the number of used disposables at any time step, not even at the cost of increasing the number of disposables at another time step. We deduce a very interesting structure of the minimum cost-per-flow-unit cycle containing $a_{pur}(k_1)$ backward.



(a) 'Red path' in the minimum cost-per-flow-unit cycle.



(b) 'Blue path' going from v_{pur} to $v_{man}(t_{sup}, 0)$.



(c) 'Blue path' from $v_{man}(t_{man}, r_{man})$ to $v_{man}(k_1, 0)$.

Figure 6.6: A minimum cost-per-flow-unit cycle containing a forward disposable arc cannot be simple, as we assumed.

Lemma 6.4.16 *Consider a flow $h^i(k_1, k_2)$ at iteration $i+1$. In a minimum cost-per-flow-unit cycle containing $a_{pur}(k_1)$ backward in $h^i(k_1, k_2)$, every arc $a_{man}(t, r)$, $a_{sup}(t, r)$ or $a_{ord}(t)$ is then backward.*

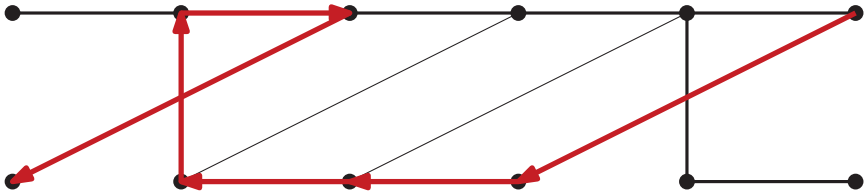
Proof:

It follows from Lemma 6.4.15 that every disposable arc in a minimum cost-per-flow-unit cycle is forward. We show for each t by recursion on r that arc $a_{sup}(t, r)$ is backward. If $a_{sup}(t, R-2)$ was forward, we would attain node $v_{sup}(t, R-1)$ whose only other adjacent arc $a_{dis}(t, R-1)$ must be forward and thus cannot be considered. Therefore $a_{sup}(t, r)$ cannot belong to a cycle, which is contradictory. We deduce that $a_{sup}(t, r)$ is backward. Suppose now that for some r arc $a_{sup}(t, r)$ cannot be backward and $a_{sup}(t, r-1)$ is forward. We would be in the same situation, where we cannot go out of node $v_{sup}(t, r)$ and hence $a_{sup}(t, r-1)$ must be backward. By extension, every ordering arc $a_{ord}(t)$ must be backward. Suppose finally that a manufacturer arc $a_{man}(t, r)$ is forward in the cycle. We assume without loss of generality that the cycle is simple, i.e. it does not contain any other cycle. There are two scenarios:

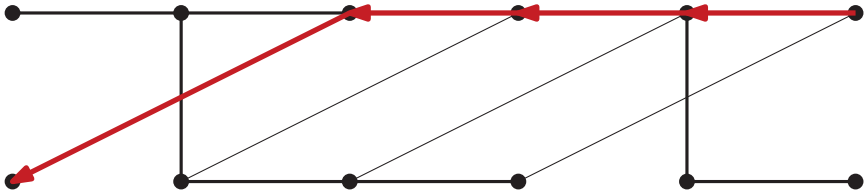
1. Every arc $a_{man}(t, r')$ for $r' \in [0, R[$ is forward. This is impossible when the cycle is simple because there is no possible path from $v_{man}(t+1, 0)$

to $v_{man}(k_1, 0)$ without going through $v_{man}(t, R-1)$ (using $a_{man}(t, R-1)$) or $v_{man}(t, 0)$ (using a disposable arc at period $t+1$).

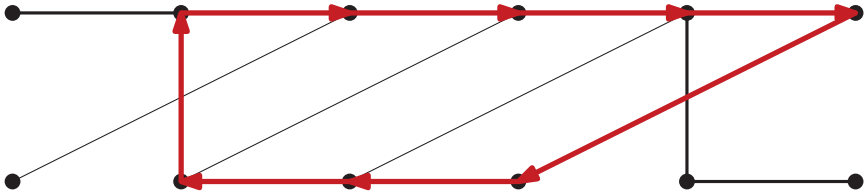
2. Every arc $a_{man}(t, r')$ for $r' \in [0, r]$ is forward as well a disposable arc $a_{dis}(t, r')$ for a $r' \in [r, L_{del}[$. As shown in Figure 6.7, there is necessarily a path in the cycle using a disposable arc $a_{dis}(t, r_2)$ as well as ordering arc $a_{ord}(t)$. These two paths form another path containing



(a) Path in the minimum cost-per-flow-unit cycle.



(b) Decomposition (2/2): simple path.



(c) Decomposition (2/2): cycle with positive cost.

Figure 6.7: Second case of having a forward manufacturer arc. Figure 6.7a represents the path (in red), while Figures 6.7b and 6.7c are its decomposition into a path and a positive cycle.

two disposable arcs, at least one of them being related to a late task. This total path is actually composed of another path and a cycle, which has positive cost by demand profitability, which contradicts the assumption that the initial cycle has minimum cost-per-flow-unit.

□

Corollary 6.4.17 *If the cycle contains $v_{man}(t, 0)$ for a period $\in]k_1, k_2[$, then it also contains either:*

1. *Arc $a_{man}(t-1, r)$ backward for all $r \in [0, R[$, corresponding to removing an idle container.*

2. *Some arcs $a_{man}(t-1, r)$ backward, an arc $a_{dis}(t-1, r)$ forward, some arcs $a_{sup}(t-1, r)$ backward and arc $a_{ord}(t-1)$ backward.*

3. *Some arcs $a_{man}(t-1,r)$ backward, an arc $a_{dis}(t-2,r)$ forward, some arcs $a_{sup}(t-2,r)$ backward and arc $a_{ord}(t-2)$ backward.*

Proof:

By lemma 6.4.16, at each node $v_{man}(t,r)$, the two only possibilities are going forward in a disposable arc or backward in a manufacturer arc. If we follow a disposable arc, we will go backward in supplier arcs and backward in an ordering arc. In this corollary, the first case corresponds to having R consecutive backward manufacturer arcs, the second case to going in a early disposable arc, and the last one to going in a late disposable arc.

□

Lemma 6.4.18 *We cannot remove more than $O(R \cdot T)$ negative cycles containing $a_{pur}(k_2)$ backward.*

Proof:

By Lemmas 6.4.15 and 6.4.16, every time we remove a negative cycle, the flow decreases on:

- some ordering arcs,
- some manufacturer arcs
- some supplier arcs
- arc $a_{pur}(k_1)$
- maybe also $a_{end}(k_2+1)$.

The flow only increases on some disposable arcs and maybe on $a_{pur}(k_2)$. Moreover, every time we remove a negative cycle, the flow on one arc goes down from begin positive to zero. Since this can happen only once for each of the above named arcs, which can decrease, we conclude that we can have at most as many iterations as the $O(R \cdot T)$ number of non-disposable arcs (and besides $a_{pur}(k_2)$) in the network.

□

Furthermore, note that the number of iterations for Lemma 6.4.18 is actually very small in practice:

- the flow in arc $a_{man}(t,r)$ is not lower than the flow in arc $a_{man}(t,r-1)$.
- once the flow in arc $a_{man}(t,r)$ is zero, the flow in arc $a_{man}(t,r+1)$ goes down to zero at the same time as the flow in arc $a_{sup}(t',r')$ incident to arc $v_{man}(t,r+1)$.

We only need to consider arcs $a_{pur}(k_1)$, $a_{ord}(t)$, $a_{sup}(t,r)$ and $a_{man}(t,0)$, which is close to one third of the arcs in the network, and around $R \cdot (k_2 - k_1 - 2)$ arcs. Moreover, we will not expect that more than half of the days of the week in average (over the periods) will use disposables, so the number in practice is halved again. Finally, we can expect that, during the first iterations, many disposable arcs are entirely filled (so with flow equal to the demand) at the same time, due to a high purchasing at period k_1, so many arcs will be emptied at the same time. The $O(R \cdot T)$ is thus a worst case upper bound.
A minimum-cost-per-flow-unit cycle can be computed with the Dijkstra shortest path algorithm in $O(R{\cdot}T{\cdot}\log[R{\cdot}T])$ time. Instead, we use Corollary 6.4.17 to compute in linear time the minimum cost-per-flow-unit cycle:

Algorithm 17: Minimum Cost-per-unit Cycle Framework

Data: k_1 and k_2
Initialize the distances Dist(k_2) and Dist(k_2+1);
for t *from* $k_2 - 1$ *to* k_1 **do**
  Dist_1(t) := cost from Dist($t+1$) being idle at period t;
  Dist_2(t) := cost from Dist($t+1$) using one more disposable for early demand at period t;
  Dist_3(t) := cost from Dist($t+2$) using one more disposable for late demand at period t;
  Dist_4(t) := cost from Dist($t+1$) using one more disposable disposable for late demand at period $t-1$, then being idle at period $t-1$;
  Dist(t) := min$\{$ Dist_i(t), $i \in [1,4]\}$;
Finish the cycle to v_{pur} with a purchasing arc at period k_1;
return the generated cycle;

Lemma 6.4.19 *Algorithm 17 computes the minimum cost-per-flow-unit cycle containing $a_{pur}(k_1)$ backward in $O(R \cdot T)$ time.*

Proof:

The algorithm iterates over $O(T)$ manufacturer nodes $v_{man}(t,0)$. For each nodes, we compute the distance from v_{pur} in $O(R)$ time by comparing three paths of length $O(R)$.

□

Theorem 6.4.20 *If Hypotheses 2.1 [Delay], 2.2 [Cost], 4.2 [Placement] and 4.4 [Distance] hold, then we can use Algorithm 17 to find negative cycles. Thus, Algorithm 16 computes a locally optimal flow $h^*(k_1,k_2)$ on $\mathcal{H}(k_1,k_2)$ in $O(R^2 \cdot T^2)$ time, for any value k_1 and k_2.*

Proof:

The initial flow $h^0(k_1, k_2)$ is computed in $O(R \cdot T)$ time by Lemma 6.4.14. There are at most $O(R \cdot T)$ iterations by Lemma 6.4.18, and each iteration done computed in $O(R \cdot T)$ by lemma 6.4.19. Theorem 6.4.13 states the correctness of the algorithm.

□

6.4.4 Extensions

Compact Networks

The adapted cycle-canceling framework is generic so that it can also be used for compact networks. In particular, we note that each of the four paths in Algorithm 17 can be transposed into the compact network.

Algorithm *Flow*.4.2

In Algorithm *Flow*.4.2, we consider up to four purchasing periods k_1, k_1+1, k_2 and k_2+1 instead of two. We recall that in network $\mathcal{H}_{2,i}(k_1, k_2)$ no container will be purchased at period k_1. Then, we use a similar initial flow $h^0(k_1, k_2)$ purchasing every container at period $k_1 + 1$. We note that arcs $a_{end}(k_2+1)$ and $a_{pur}(k_2+1)$ cannot be non-empty at the same time. Then, we use the same negative cycle algorithm, since exactly as for $a_{end}(k_2+1)$ and $a_{pur}(k_2)$ we only have to look for a negative cycle containing arcs $a_{man}(k_2, r)$ for $r \in [0, R[$. We can also use compact networks.

Algorithm *Flow*.5.1

Just like Algorithm *Flow*.4.1, the networks used in Algorithms *Flow*.5.1 are set between two placements k_1 and k_2. Nevertheless, we satisfy demands at periods k_1 and $k_1 + 1$ instead of k_2 and $k_1 + 2$. We can also start with a policy using zero disposable and purchasing every containers at period k_1. The invariant has to be updated so that every negative cycle contains arc $a^{5.1}_{end}(k_2)$ backward. Every negative cycle contains either $a^{ord}_{pur}(k_1)$ or $a^{idle}_{pur}(k_1+1)$. We note that the distance of $v_{man}(k_2+1, 0)$ is computed from $a^{5.1}_{end}(k_2)$ and the distance of $v_{man}(k_2+1, 0)$ is computed from $v_{man}(k_2+1, 0)$ with idleness at period k_2, since there is no demand to fulfill at period k_2. Every result can then be extended and the computation of a negative cycle follows the same pattern as Algorithm 17, besides a special treatment at periods $k_2 + 1$, $k_1 + 1$ and k_1. We can also use compact networks.

Algorithm *Flow*.5.2

The networks for Algorithm *Flow*.5.2 are nearly identical to the networks for Algorithm *Flow*.5.1, as we only replace the manufacturer arc $a_{man}(k_2, 0)$

with arc $a^{5.2}_{end}(k_2)$, and $a^{5.1}_{end}(k_2)$ with $a^{5.2}_{end}(k_2+1)$. For the invariant, we require every negative cycle to contain either $a^{ord}_{pur}(k_1)$ or $a^{idle}_{pur}(k_1+1)$ backward, and either $a^{5.2}_{end}(k_2)$ or $a^{5.2}_{end}(k_2+1)$ backward. In addition, the distance of node $v_{man}(k_2,0)$ is computed directly from $a^{5.2}_{end}(k_2)$, since there is no demand to fulfill at period k_2. Every other result can be extended, and we can also use compact networks.

Conclusion

Theorem 6.4.21 *The adapted cycle-canceling algorithm can be used for both extended and compact network, and all flow algorithms. Flow.*4.1*, Flow.*4.2*, Flow.*5.1*, Flow.*5.2*. The worst case time complexity is* $O(R^2 \cdot T^2)$.

6.5 Simulations

We compare the running time of the four flow-based algorithms in six different settings. We consider the two representations, namely extended and compact. Moreover, to compute a minimum linear-cost flow we use either the enhanced capacity-scaling algorithm (E.C.S), the adapted cycle-canceling algorithm (A.C.C.) or a standard capacity scaling algorithm (Std.C.S).
The results on Tables 6.6 and 6.7 are average running times over $N = 10$ simulations with randomly generated data. We denote by $\mathcal{U}(m, M)$ the discrete uniform distribution with minimum value m and maximum value M, and by $\mathcal{B}(n, p)$ the binomial distribution with n experiments and probability p of success for each experiment. We also use operations on the distributions, so that $10 \cdot \mathcal{B}(10, 0.5) + 5$ is a distribution taking values 5, 15, 25, ..., 105.

- $T \to \mathcal{U}(15, 30)$, $R \to \mathcal{U}(3, 7)$.
- L_{del} follows a uniform distribution taking any odd value in $[3, R]$.
- For each $r \in [0, R[$, the demand distribution follows an additive autoregressive model[4]. Demand $D(0, r)$ follows distribution $\mathcal{U}(50, 100)$. At time $t > 0$, demand $D(t, r)$ follows the sum of distribution $\mathcal{U}(-10, 20)$ and the distribution $D(t-1, r)$.
- $\forall t, \forall r, C_{man}(t, r) \to \mathcal{U}(1, 2)$.
- $\forall t, \forall r, C_{sup}(t, r) = 6$.
- $\forall t, \forall r, C_{dis}(t, r) = 100$.
- $\forall t, C_{setup}(t) \to 50 \cdot \big(2 + \mathcal{B}(4, 0.8)\big)$.

[4]This is a common model used to describe the evolution of a variable. It also enters into the famous martigale model of forecast evolution framework [45]

Table 6.6: Average running times (in seconds) of the flow algorithms for each variant. In this simulation, the setup cost follow distribution $10000 \cdot \big(1 + \mathcal{B}(20, 0.4)\big)$ at every period.

	Algorithm	*Flow*.4.1	*Flow*.4.2	*Flow*.5.1	*Flow*.5.2
	Cost	551529	551529	551529	551529
E.C.S.	Extended	18.7	70.9	20.8	51.1
	Compact	8.7	32.1	9.7	22.2
Std C.S.	Extended	7.4	29.6	8.6	20.5
	Compact	5.2	20.0	6.2	14.5
A.C.C.	Extended	0.16	0.62	0.1	0.25
	Compact	0.08	0.3	0.06	0.16

Table 6.7: Average running times (in seconds) of the flow algorithms for each variant. In this simulation, the setup cost follow distribution $5000 \cdot \big(1 + \mathcal{B}(20, 0.4)\big)$ at every period.

	Algorithm	*Flow*.4.1	*Flow*.4.2	*Flow*.5.1	*Flow*.5.2
	Cost	552124	541170	549596	538475
E.C.S.	Extended	25.5	98.8	29.1	65.2
	Compact	11.0	44.4	12.7	29.2
Std C.S.	Extended	10.2	40.6	11.8	25.9
	Compact	6.9	27.1	8.4	18.9
A.C.C.	Extended	0.20	0.764	0.13	0.28
	Compact	0.1	0.36	0.08	0.17

First of all, the experiments show that the standard capacity scaling algorithm is in average twice as fast as the enhanced capacity scaling algorithm. While we may not have implemented the later algorithm as efficiently as it should be possible, we believe that this result is not unexpected. The enhanced capacity scaling algorithm requires to separate the nodes into trees and merge some trees after possibly every shortest path algorithm. This algorithm hence takes a lot of time to ensure that at each iteration enough flow is sent from a source to a sink. Nevertheless, in our networks, the sinks have in general a close excess value, so there should be no need to use the enhanced capacity scaling algorithm. Furthermore, the enhanced capacity scaling algorithm systematically divides the Δ flow of the shortest paths by two, resulting in decimal flow quantities. Meanwhile, our standard capacity scaling algorithm starts with a power-of-two initial Δ flow, thus it only considers integral flows. We believe that this difference is the reason for slowing down the enhanced capacity scaling algorithm.

Secondly, the adapted cycle-canceling algorithm is approximately 75 times faster than the standard capacity scaling algorithm for this simulation data.

This shows that the number of shortest paths we compute is fairly low, and that this alternative algorithm is very well adapted to our network.
Thirdly, it turns out that the use of the compact network representation halves the running time of all three minimum linear-cost flow algorithms. It is interesting to see that we keep that speed increase from the compact network representation even when using the cycle-canceling algorithm, despite the computation of the path being nearly the same for both network formulations. We recall that the minimum linear-cost flow algorithms can be computed separately using different threads, so the computational improvement from using several threads naturally stacks up with the running time decrease from using a compact network formulation and the adapted cycle-canceling algorithm.
Finally, compared to their initial version described in Chapters 4 and 5, our three algorithmic improvements speed up every flow algorithm by a factor of 300. The final complexity is below half of a second when considering a container purchasing plan over half a year, with $T = 30$ and $R = 7$. This complexity is very satisfying for our application, where we consider a time horizon of one year with five working days per week.

6.6 Outlook

In this chapter we have analyzed the solution quality and the running time of our deterministic algorithms. Firstly, we compare the quality of the policies described in Chapters 4 and 5. It turns out that the policy forbidding disposables is noticeably more costly than our policies allowing them. This shows that even in a deterministic setting it is meaningful to allow disposables. Moreover, the flow algorithms from Chapter 4 are in practice nearly as good as the flow algorithms from Chapter 5, despite not being designed for a general demand pattern. The former algorithms are only performing poorly when the demand is not steadily increasing and when there is a very low demand at the desired purchasing time. In this case, Algorithms $Flow$.4.1 and $Flow$.4.2 will use a limited fleet size.
The flow-based algorithms are nearly always optimal but are a little slow, so we proceeded to improve their experimental running time. The first improvement uses several threads, as we experimentally highlight that the algorithms are easily parallelizable. Our 2-core machine hence computes a policy twice as fast using two or three threads. The second idea is to formulate network $\mathcal{G}$ in a more compact way, only using 60% to 80% as many nodes and arcs. This method also halves the running time of the flow algorithms. However, it systematically excludes a cost from the system so we have to compute it separately. Finally, we developed a cycle-canceling algorithm to compute minimum linear-cost flows on our specific networks. Compared to the enhanced capacity scaling algorithm, this approach is around 70 times

faster. Using these three algorithmic improvements, we reduce the total running time from nearly one second to hundreds of milliseconds for data of average size: $T = 30$, $R = 7$.

Part II

Stochastic Study

Chapter 7

Strategies for the Stochastic Model

We consider the stochastic version of the container purchasing problem, which we denote by ($SCPP$). Every demand is now stochastic and its distribution is fixed at the beginning of the time horizon. In a previous paper [55], we study $SCPP$ using a Markov decision process framework, and extend two basic strategies from the stochastic lot-sizing literature. This chapter develops this earlier work and completes our framework with two hybrid strategies.
In Section 7.1, we review the related literature on stochastic problems. Section 7.2 describes our Markov decision process framework and defines the objective function according to four different strategies. Sections 7.3, 7.5 and 7.6 repectively present the online strategy, the offline strategy and the two hybrid strategies, which we call the quasi-online and the quasi-offline strategies. In Section 7.4, we prove that the cost functions of our problem are $L^\natural$-convex for the online policy if we remove the setup costs. Our policies are experimentally evaluated in Section 7.7. Finally, Section 7.8 summarizes our study and states some extensions of the model.

7.1 Literature Survey

Similar to the deterministic setting, a lot of research has been done on stochastic inventory control problems, but there is little literature on determining the optimal fleet size. To the best of our knowledge no article searched a purchasing plan of containers in a ramp-up scenario.
We restrict our review to periodic review systems, where decisions can only made at specific points in time. It is usually assumed that the time periods between two consecutive points have the same length and represent a time unit. We divide the existing related literature into three categories: stochastic lot-sizing problems (SLSP), inventory control problems with lost-

sales (ICPL) and empty container repositioning problems (ECRP). Just like in the stochastic setting, the lot-sizing problems represent the literature with a setup cost for purchasing and the empty container repositioning problems consider returnable items in a closed-loop supply chain. Inventory control problems have neither of these properties, but the used methods are relevant for our work.

7.1.1 Stochastic Lot-Sizing Problems

Following the deterministic lot-sizing model of Wagner and Within [128], researchers realized the importance of taking into account the uncertainty in lot-sizing decisions [131]. We refer to Aloulou et al. [4] for a general literature on stochastic lot-sizing problems. Early, Silver [106] as well as Askin [6] proposed some heuristic algorithms under a simple demand distribution pattern. In particular, Silver [106] extends the Silver-Meal heuristic [107] to a stochastic setting. We recall that the Silver-Meal heuristic consists of approximating at each placement the purchase size and the time of the next placement. Both papers adapt a deterministic heuristic to a stochastic environment. Several other papers deal with uncertain demands by determining a safety stock buffer (see [4]).
A fundamental contribution is by Bookbinder and Tan [11]. They proposed a classification of strategies for SLSP with a fixed time horizon into three groups. Firstly, the *dynamic uncertainty* strategy consists in computing a solution in an online manner, i.e. taking every decision as late as possible to make the best choice given the latest information. Secondly, the *static uncertainty* strategy consists in deciding at the beginning of the time horizon on the placements and the purchase quantities. The algorithm is thus offline and may not be as efficient as a dynamic uncertainty strategy. However, the planning does not have to be constantly updated, which offers more stability and less *nervousness* to the solution. Finally, the *static-dynamic uncertainty* strategy is a compromise where the placements are fixed at the beginning of the time horizon but the purchase quantities are decided online, depending on the left-over inventory. However, in a context of container management, the static-dynamic approach has no a priori advantage compared to the static uncertainty strategy. The authors of [11] consider a service level constraint instead of a penalty cost for not meeting a demand. We note that the Silver heuristic [106] which is frequently used in practice does not correspond to any of their three strategies. In this thesis, we call *policy* a decision function and *strategy* the set of policies following the same decision pattern. Given a strategy, we say that a policy is *optimal* if it minimizes the cost over all other policies following this strategy. We can hence call a static uncertainty policy optimal even though is has higher cost than a dynamic uncertainty policy. We often refer to a policy following the dynamic uncertainty strategy as an *online policy* and to a policy following

the static uncertainty strategy as an *offline policy*.

The dynamic uncertainty strategy has a very abundant literature which overlaps traditional inventory control problems without setup cost. We present the online strategy together with inventory control problems in the next subsection.

Literature on the static uncertainty strategy is rather scarce. The reason is that this strategy usually does not generate efficient policies, because it cannot adapt to the demand realizations. In contrast, the static-dynamic strategy found much more success as we can raise the inventory position to a satisfying level, so that the stock after ordering is loosely dependent to the previous demand realizations. For the static uncertainty model, Sox [115] considers a rolling horizon framework with backlogging cost. Haugen [42] use a *progressive edging* meta-heuristic to solve SLSP. Vargas [125] as well as Piperagkas [87] also study the static uncertainty strategy of SLSP.

In the following, we present papers on the static-dynamic uncertainty model. Pujawan [92] considers two traditional heuristic algorithms for deterministic lot-sizing problems and analyze their properties and behavior under demand variability. Pujawan and Silver [93] propose two heuristic static-dynamic algorithms for SLSP. They show through numerical experiments that these heuristics perform better than the usual dynamic-uncertainty policy (s, S), ordering up to stock S whenever the stock falls below value s. Tarim and Kingsman [117] consider the static-dynamic strategy of Bookbinder and Tan [11] and formulate a mixed integer linear program. They also assume a minimum service level constraint instead of a penalty cost. They allow the linear cost function to be dynamic and show that their solution is an improvement over earlier work [11]. In a following paper [118], the same authors replace the minimum service level with a backlogging cost. Again, the problem is solved using a mixed integer linear program, and the authors look for a *replenishment cycle policy* (R_t, S_t). The traditional base-stock policy for stationary demand and zero setup cost consists of ordering at each period to raise the stock up to a fixed level S. Such a policy is optimal if there is no lead time. When there is a setup cost, there are four main classes of parametric policies in the literature, as summarized on Table 7.1. Firstly, two policy classes fix an ordering interval and the two others order whenever the stock falls below a threshold. Secondly, for two strategies we order the same quantity at every order and on the two others we order up to a certain level. The (R, S) policy orders up to stock S every R periods. A

	Every R periods	When stock falls below s
Order size q	(R, q) policies	(s, q) policies
Order up to S	(R, S) policies	(s, S) policies

Table 7.1: Four simple classes of ordering policy.

straightforward generalization for non-stationary demand disributions is to make the parameters R and S depend on t. Consequently, when ordering at period t, we raise the inventory up to stock level S_t and set the next order time at period $t + R_t$. This policy follows the static-dynamic uncertainty strategy because every parameter R_t and S_t is decided at the beginning.
The study of Özen et al. [86] is maybe the most similar paper to our stochastic study. They compute a static-dynamic solution to the stochastic lot-sizing problem with lost-sales by coupling a Markov decision process with a simple dynamic program. They then approximate the stock we have left at ordering time using two intuitive heuristics and show experimentally that the results are close to optimal while drastically decreasing the running time. Rossi et al. [97] use constraint programming to compute the optimal (R_t, S_t) policy of the SLSP with penalty cost. The same authors improve the running time of their constraint programming approach using filtering methods. They use this new method to solve the problem under a minimum service level constraint in [116] and under a backlogging cost in [100]. In Rossi et al. [98], they extend the model to stochastic lead times. Rossi et al. [99] compute an optimal (R_t, S, t) policy using state space aggregation and state space augmentation in a static-dynamic model of SLSP where unmet demand is backlogged, hence without any service level constraint. In [96], they propose a generalized framework to compute near-optimal solutions of the SLSP following a static-dynamic strategy, and including service constraints, backlogging or lost-sales. Tunc et al. [120] reformulate the mixed integer program of [11] for the static-dynamic strategy of the SLSP. They show that their algorithm is very fast for large problem instances.

7.1.2 Inventory Control with Lost-Sales

We now give an overview of the literature with online stochastic inventory control problems. The objective is to compute online policies with or without setup cost. In particular, an online lot-sizing problem is not any different from an inventory control problem with setup cost.
We recall that there are two main ways of dealing with stock-out, namely backlogging and lost-sales. The distinction between lost-sales and backlogging only makes sense for positive lead time. Our container management problem has a lost-sales behavior. We refer to Bijvank and Vis [10] for a review of lost-sales problems in a different setting than ours, as in continuous review models and perishable products, for example.

Lost-Sales and Backlogging Models

It is well known in the literature (see for exmaple [108, 82, 143]) that traditional inventory control problems with backlogging, stationary demand and no setup-cost is very easy to solve. We define the *inventory position* as the

inventory level plus the sum of outgoing orders. For this class of problems with backloging, the optimal ordering policy is a function with the form $S_t(x) := S - x_t$, where x_t is the inventory position at ordering and S is the optimal inventory position. When we add setup costs, the optimal order size is characterized by two values (s, S), where the inventory levels is raised up to S whenever it falls below a threshold s. We refer to *base-stock policies* as those policies, defined by an inventory position S with or without the ordering threshold s.

Inventory control problems with lost-sales are more complex to solve. One of the earliest work is from Karlin and Scarf [62], who showed in particular that the optimal policies are not base-stock policies when the lead time is positive. The authors have also modeled the problem as a general Markov decision process and analyzed some basic properties when the lead time equals one period. Morton [80] has generalized basic results from [62] to any positive integral lead time, and developed efficient bounds on the optimal policy. Later, Morton [81] proposed some myopic policies and experimentally analyzed their performance. Zipkin [142] experimentally compares different policies in inventory problems with lost-sales and shows that simple myopic policies perform much better than the best base-stock policy. In other words, it is better to minimize the cost for the coming period only rather than considering the best inventory position. The biggest issue with myopic policies for many applications is their long computation time. Indeed, myopic policies separately compute the order size for each state. Instead, the base-stock policies map the order policy using a parametric function, which can be efficiently approximated. A good alternative to base-stock policies is the class of *restricted base-stock policies* introduced by Johansen and Thorstenson [58]. Restricted base stock policies only differ from simple base-stock policies by having a maximum order size R, and have been shown to perform very well in practice. Zipkin [142] considers a similar alternative called the *vector base-stock* policy with a single parameter representing a service level. The order size is set such that the inventory position ensures approximately this service level over the coming periods.

Several papers further analyze the performance of optimal base-stock policies in lost-sales problems. Janakiraman et al. [57] compare simple inventory problems with positive lead time under backlogging and under lost-sales. Huh et al. [50] prove that any optimal base-stock policy is asymptotically optimal for lost-sales inventory systems when the shortage cost gets large. This result can be interpreted as follows. On one hand, restricted base-stock policies are close to optimal. On the other hand, a simple base-stock policy with parameter S only orders a different quantity than most restricted base-stock policies with parameter (S, R) when the inventory position is low. When the shortage cost is large, the best restricted base-stock policies keep a high stock to avoid stock-out. It is thus nearly equivalent to a base-stock policy. In another work, Huh et al. [49] propose an adaptive algorithm

to compute the optimal base-stock policy when the demand distribution is stationary but unknown. The algorithm is shown to converge very fast.
In practice, both the backlogging models and lost-sales models require a long computation time, especially when the demand is not stationary. Therefore, some researchers look for efficient approximation methods. Levi et al. [70] introduce the class of *dual-balancing* policies. In dual balancing policies, the expected holding cost, shortage cost and purchasing cost are equal. These strategies have a provable performance guarantee. In a setting with backlogging, immediate ordering and zero lead time, the expected cost of the dual-balancing policy is at most twice the expected cost of the best online policy. Nonetheless, this bound holds for the worst case scenario and the performance of the policy is much closer to optimal in most cases. Levi et al. [69] adapt this policy to lost-sales models and prove that the policy has the same performance guarantee. We discuss these policies in the next Chapter.

Convex Optimization

Several researchers pay attention to convexity properties. Convex problems are easier to solve, since we can use a binary search to compute an optimal order size. Moreover, convexity may ensure the convergence to optimality for several meta-heuristics. Janakiraman and Muckstadt [56] derive properties of optimal policies in a stochastic inventory control system with lost-sales and fractional lead time. Fractional lead times correspond to transportation times shorter than one week in our application. In particular, they prove the convexity of the cost following an ordering decision with respect to the stock at ordering. Furthermore, they show that the slope of this cost function is in $[-1, 0]$ and give simple upper and lower bounds of the optimal order size. In contrast, our supplier and manufacturer holding costs are incurred not only at the end of each week, but also every day. Closer to us is the work of Chiang [16]. The author considers an inventory control problem with replenishment every R time steps and a lead time smaller than R. He writes a dynamic program and proves for the lost-sales setting that the cost function is convex. Moreover, he shows that the derivative of the optimal order size is a convex function of the stock at hand. Halman et al. [40] solve a general stochastic inventory control problem using dynamic programming. Moreover, they prove the convexity of the cost functions and derive a *fully-polynomial time approximation scheme* (FPTAS): for every ε, the algorithm is polynomial in ε^{-1} and is an ε-approximation, so that the expected cost of the generated policy is at most $(1 + \varepsilon)$ times the expected cost of the optimal policy. Halman et al. [39] describe a framework to create FPTAS, in particular for stochastic inventory control problems.
More recently, work has been done on convex problems with larger lead times. Indeed, most of the previously cited papers consider a short lead

time such that we only need one parameter to represent the state space. When the lead time is larger than the review period, the traditional convexity property does not hold anymore. Instead, we use $L^\natural$-convexity (L-natural convexity), which is a generalization of convexity introduced by Murota[83]. Zipkin [141] characterizes optimal inventory control policies in a system with lost-sales. He shows that the cost functions in a Markov decision process framework are $L^\natural$-convex for a specific state space. In Section 7.4, we present these results as described by Simchi-Levi et al. [108] and use them to show the $L^\natural$-convexity of the cost functions. Chen et al. [14] use the $L^\natural$-convexity properties from [141] and deduce a pseudo-polynomial approximation scheme similar to the FPTAS in [40]. Their paper adapts the work of Halman et al. [40] to multi-dimensional state, decision and demand spaces. However, their approximation scheme is only pseudo-polynomial, and it is an open question whether a fully polynomial approximation scheme exists for their problem. Huh and Janakiraman [48] extend the work of Zipkin [141] to serial supply chains. Our work in this chapter adapts these results to container management problems without setup costs.

Further Research on Online Stochastic Lot-Sizing

Guan et al. [35] use a branch-and-cut algorithm on a tree data structure to generate a dynamic strategy for the SLSP, and extend some properties of the optimal solution in the deterministic case. Their work is extended by Guan and Miller [36], who present the equivalent of the fundamental zero inventory property of Wagner and Within [128] to a stochastic environment. Nevertheless, the SLSP problem is proven to by NP-complete by Halman et al. [40], so one can only create an algorithm which is polynomial in the size of the scenario tree structure. Li et al. [72] consider a SLSP with remanufacturing and solve the dynamic model using stochastic dynamic programming. Wagner [129] considers two online lot-sizing problems with the additional restriction that the manager has no information on the future demands and on the length of the planning horizon. He compares the cost of the algorithms to offline algorithms having full knowledge of this information.

7.1.3 Stochastic Container Management

Stochastic container management problems extend deterministic container management problems to stochastic data. The uncertainty may concern various parameters like demand value, lead time, container deterioration and so on. We restrict our literature review to container management problems where the only stochastic parameter is the periodic demand value. We note that most articles present a *container repositioning problem* without a purchasing option. In addition, the repositioning time is usually assumed immediate and researchers look for an optimal base-stock policy (s, S).

In 1987, Dejax and Crainic [19] presented a review of the stochastic fleet management problems, including both truck management and container management. Crainic et al. [18] as well as Cheung and Chen [15] model a general container management problem as a two-stage stochastic integer program.

Container management problems are very complex to solve. They are frequently modeled as Markov Decision Processes. Powell [91, 90] recommends to use approximate dynamic programming to efficiently solve logistics problems. We present and discuss approximate dynamic programming in Chapter 8 as an alternative and a possible extension of our work in this chapter. Lam et al. [66] applied approximate dynamic program using *temporal differences* on a simple maritime container repositioning problem. Their model includes neither positive lead time nor container purchasing.

Li et al. [74, 73] study an empty container repositioning problem in the maritime industry, first at a single port and then over multiple ports. They consider the problem from a theoretical aspect and prove that the optimal policy at each port is a *threshold policy* under zero lead time and stationary demands. A threshold policy is defined by two variables l_p and u_p for each port p. The stock at any port p is raised up to l_p if it was below it, and decreased down to u_p if it was above it. A similar result is proven by Song [110] for a single port, and by Song and Earl [112] for a two depots system. Zhang et al. [139] consider a similar container repositioning problem with lost-sales. For a single port, they show that the optimal repositioning policy is a threshold policy and compute it in polynomial time. For multiple ports, they approximate the best threshold policy using a similar algorithm. For a single port and infinite fleet size, Song [109] computes an optimal threshold policy in a continuous review inventory model.

Dong and Song [22] consider a joint container fleet sizing and repositioning problem in a system between several maritime ports under stationary demands and zero lead time. The container fleet sizing can be defined as a container purchasing of containers at every port during the first time period, so there is no initial repositioning. They consider a liner shipping process where the ships transporting containers follow some specific routes. They model a stochastic program and solve it using a gradient-based method. They extend their work in [23] to consider further inland transportation. Lee et al. [68] consider a similar study as Dong and Song [22], and also propose a gradient-based method to compute a threshold policy and a container fleet size.

Song and Zhang [113] prove that without repositioning delay a two level threshold policy is optimal for a single port, assuming an infinite external fleet size. Song and Zhang [114] consider a container repositioning problem with positive lead times. The repositioning decision is a rule-based inventory control policy. Since they consider transportation delays, their work is close to ours. However, they consider a stationary demand pattern and measure

the steady-state performances of their heuristic threshold policy.
Erera et al. [24] study container repositioning in the chemical industry. They extend their previous paper [25] with deterministic demands to uncertain demands, using a a robust optimization with budget: Every single demand has to be fulfilled. Whenever the stock of empty containers is lower than the demand, empty containers are immediately imported at a cost. The repositioning before the demand comes out is computed to minimize the cost under the contraint that the immediate repositioning cost after the demand realization does not exceed a budget constraint.
Yi [135] and Yin [136] solve in their PhD theses a stochastic container management problem using meta-heuristics, respectively the Compass method and a progressive edging algorithm. In [135], the author focuses on threshold policies in a container repositioning problem with stationary demands. The container fleet size is a decision to take at the beginning of the time horizon. The maximum necessary fleet size is the sum of the optimal threshold values. However, they show that this maximum fleet size is usually not the optimal fleet size. Therefore, in our container management problem with increasing demand, it is clearly not the case either.
For further literature results on container management problems, we refer to Dong and Song [111].

7.2 Problem Description

7.2.1 Notations and Strategies

Contrary to the deterministic setting, we denote now the demand by $D_{t,r}$ instead of $D(t,r)$. Furthermore, the ordering delay L_{ord} is assumed positive. We assume that the demand cannot exceed a value $D_{\max}$.
We call *order size* β_t the number of empty containers sent from the manufacturer to the supplier before $(t,0)$, and denote by α_t the number of containers purchased before ordering. At time (t,r), the sequence of events is:

1. If $r=0$, the manufacturer purchases α_t containers, then the supplier orders β_t empty containers.

2. Demand $D_{t,r}$ occurs and is satisfied using the available packages.

3. Outgoing containers arrive to their respective locations:

   - The containers used for demand at time $(t,r)-(L_{del}-1)$ are added to the manufacturer stock.
   - If $r=L_{ord}-1$, β_t containers are added to the supplier stock.

At the beginning of each period t, we denote by X_t the supplier stock, by Y_t the manufacturer stock and by Z_t the number of outgoing full containers.

We call *container fleet size* U_t the number of containers in the system before purchasing. By Hypothesis 2.1 [Delay], the ordering delay is smaller than one period so there is no outgoing order at time $(t, 0)$. Thus:

$$U_t = X_t + Y_t + Z_t. \tag{7.1}$$

We denote by $U^\alpha(t)$ the number of containers in the system after purchasing at period t. We assume that at the beginning of the time horizon, there is no container in the system.
Our objective is to find a purchasing and ordering policy minimizing the expected cost over the whole time horizon. Container ordering is an operational decision because it should to be done dynamically depending on the supplier stock. Thus, we make the order decision β_t during period t. Meanwhile, we see container purchasing as a tactical decision because a purchasing decision should look into the future to avoid unnecessary purchasing in the near future, due to the setup cost. We consider four different strategies:

1. An *online strategy*, where the purchase size α_t is decided at period t.
2. An *offline strategy*, where α_t is decided at period 0.
3. A *quasi-offline strategy*, where we decide on the purchasing times at period 0, i.e. whether $\alpha_t = 0$ or $\alpha_t > 0$. At period t we decide on the actual purchase size if we decided that $\alpha_t > 0$.
4. A *quasi-online strategy*, where we decide at each placement the number of purchased containers and the time of the next placement.

The online, offline and quasi-offline strategies extend respectively the dynamic, static and static-dynamic strategies from Bookbinder and Tan [11]. The quasi-online strategy corresponds to the Silver-Meal heuristic [106]. We call *hybrid strategies* the quasi-offline and the quasi-online strategies.
The quasi-online strategy is a meaningful alternative to the online strategy because a tactical decision like the container fleet size should be planned ahead of time. Instead, an online policy waits until the last moment to decide whether or not to purchase new containers at a period.
On the other hand, it is not clear at first sight whether a quasi-offline policy is a significant improvement over the offline policy. In traditional inventory control problems, the static-dynamic strategy is a great compromise between the static and the dynamic policies. Since the static policy fixes the purchase sizes right from the start, its performance is heavily influenced by the demand realizations. Contrary to the static-dynamic policy, it has no way of correcting the inventory level after facing a surge or a drop of demand. In a container purchasing problem however, the offline policy should not be as sensitive to the demand realizations, since every container we send to the supplier will return after a couple of periods. It is hence meaningful to compare the performance of an offline policy to a quasi-offline policy.

7.2.2 Problem Complexity

Lemma 7.2.1 ***(Halman et al. [40])*** *The problem of computing the convolution of N independent random variables is #P-hard with respect to N. In particular, this problem is equivalent to computing the distribution of the sum of N independent random variables.*

Proposition 7.2.2 *Even under stationary cost functions, the container production planning problem is #P-hard with respect to R for each of the four strategies.*

Proof:

Suppose that the container purchasing costs $C_{setup}(t)$ and $C_{cont}(t)$ are zero and the time horizon consists in a single period: $T = 1$. Thus, every decision is made at the beginning of the time horizon, and any policy follow at the same time any of the four strategies. The objective is to find an order quantity minimizing the cost over R time steps of demand. This is equivalent to computing the distribution of the sum of R independent random variables. By Lemma 7.2.1 this problem is #P-hard with respect to R.

□

7.3 Online Strategy

We model the online problem as a Markov decision process over $R \cdot T$ time steps, where the state at time (t, r) is noted $\boldsymbol{S_{t,r}}$. The decision $\boldsymbol{\gamma_t}$ at period t is:

$$\boldsymbol{\gamma_t} := [\alpha_t, \beta_t] \tag{7.2}$$

In the stochastic setting, we use capital letters for random variables and lower case letters for their realization. The realizations of $\boldsymbol{S_{t,r}}$, $X_{t,r}$, $Y_{t,r}$, $Z_{t,r}$, U_t, $U^\alpha(t)$ are thus written $\boldsymbol{s_{t,r}}$, $x_{t,r}$, $y_{t,r}$, $z_{t,r}$, u_t, $u^\alpha(t)$. Moreover, we use a bold font to denote vectors, as opposed to a normal font for scalars. We denote by $\mathbb{P}_{t,r}(d) := \mathbb{P}(D_{t,r} = d)$ the probability that variable $D_{t,r}$ takes value d. We only need three parameters in $\boldsymbol{S_{t,r}}$:

$$\boldsymbol{S_{t,r}} := [X_{t,r},\ Y_{t,r},\ Z_{t,r}]. \tag{7.3}$$

Since the decisions are only taken at times $(t, 0)$, we define $\boldsymbol{S_{t,0}} := [X_t, Y_t, Z_t]$ as our main states, while states $\boldsymbol{S_{t,r}}$ for $r > 0$ are only transition states used to avoid the computation of R simultaneous demand variables, which would take a lot of time as shown by Lemma 7.2.1. In the following, we firstly define the variables $X_{t,r}$, $Y_{t,r}$ and $Z_{t,r}$, then write the equations corresponding to the transition from a state $\boldsymbol{s_{t,r}}$ to the next state $\boldsymbol{s_{t,r+1}}$.
Consider $t \in [0, T[$ and $r > 0$. We define $X_{t,r}$ as the supplier stock at time (t, r). We define $Y_{t,r}$ as the *ensured manufacturer stock* at time (t, r)

for time $(t+1,0)$, that is, the current manufacturer stock plus the number of full containers which left the supplier before time (t,r) and will arrive between times $(t,1)$ and $(t+1,0)$. We put the leftover information relative to the outgoing number of containers into variable $Z_{t,r}$. For $r \in [1, L_{ord}-1]$, $Z_{t,r}$ equals the order size β_t. Between time steps L_{ord} and $R - L_{del}$, we do not need any other information than $X_{t,r}$ and $Y_{t,r}$, so we can set $Z_{t,r} = 0$. Finally, for $r \in]R-L_{del}, R[$, we count the number of outgoing full containers which will arrive after the next ordering time $(t+1,0)$. Note that X_t, Y_t, Z_t correspond to the definition of $X_{t-1,R}$, $Y_{t-1,R}$, $Z_{t-1,R}$, so our notations are consistent. Let $\Delta_{t,r}$ be the number of containers used for demand $D_{t,r}$. Under Hypothesis 2.2 [Cost], we have:

$$\Delta_{t,r} := \min\{D_{t,r}, X_{t,r}\} \tag{7.4}$$

Consider state $\boldsymbol{s_{t,r}} = [x_{t,r}, y_{t,r}, z_{t,r}]$ at time (t,r). The demand realization $d_{t,r}$ induces a number $\delta_{t,r} := \min\{d_{t,r}, x_{t,r}\}$ of full containers sent from the supplier to the manufacturer. In addition, if $r = 0$ we take the decision $\boldsymbol{\gamma_t} = [\alpha_t, \beta_t]$. Under Hypothesis 2.2 [Cost], the next state at time $(t, r+1)$ is computed using the following equations:

- For $r = 0$:

$$\boldsymbol{s_{t,1}}[\boldsymbol{s_{t,0}}, \boldsymbol{\gamma_t}, d_{t,0}] = [x_{t,0} - \delta_{t,0};\ y_{t,0} + \alpha_t - \beta_t + z_{t,0} + \delta_{t,0};\ \beta_t] \tag{7.5}$$

- For $r \in [1, L_{ord} - 2]$:

$$\boldsymbol{s_{t,r+1}}[\boldsymbol{s_{t,r}}, d_{t,r}] = [x_{t,r} - \delta_{t,r};\ y_{t,r} + \delta_{t,r};\ z_{t,r}] \tag{7.6}$$

- For $r = L_{ord} - 1$:

$$\boldsymbol{s_{t,r+1}}[\boldsymbol{s_{t,r}}, d_{t,r}] = [z_{t,r} + x_{t,r} - \delta_{t,r};\ y_{t,r} + \delta_{t,r};\ 0] \tag{7.7}$$

- For $r \in [L_{ord}, R - L_{del}]$:

$$\boldsymbol{s_{t,r+1}}[\boldsymbol{s_{t,r}}, d_{t,r}] = [x_{t,r} - \delta_{t,r};\ y_{t,r} + \delta_{t,r};\ 0] \tag{7.8}$$

- For $r > R - L_{del}$:

$$\boldsymbol{s_{t,r+1}}[\boldsymbol{s_{t,r}}, d_{t,r}] = [x_{t,r} - \delta_{t,r};\ y_{t,r};\ z_{t,r} + \delta_{t,r}] \tag{7.9}$$

Lemma 7.3.1 *Under Hypotheses 2.1 [Delay] and 2.2 [Cost], we have at each time (t,r):*

- $0 \leq x_{t,r} \leq (R + L_{ord}) \cdot D_{\max}$
- $0 \leq y_{t,r} \leq (R + L_{ord} + L_{del}) \cdot D_{\max}$

- $0 \leq z_{t,r} \leq L_{del} \cdot D_{\max}$
- $0 \leq \alpha_t \leq (R + L_{ord} + L_{del}) \cdot D_{\max}$
- $0 \leq \beta_t \leq R \cdot D_{\max}$

Moreover, an upper bound of the optimal container fleet size is:

$$U_M := (R + L_{ord} + L_{del}) \cdot D_{\max} \leq 2 \cdot R \cdot D_{\max} \tag{7.10}$$

Proof:

The maximum demand between two order arrivals is $R \cdot D_{\max}$. Therefore, whenever we order more than $R \cdot D_{\max}$ containers, some containers will be left in the supplier stock before the next order arrival, which is never profitable by Hypothesis 2.2 [Cost]. We deduce: $\forall t,\ \beta_t \leq R \cdot D_{\max}$. Likewise, if the inventory position is above $(R + L_{ord}) \cdot D_{\max}$, the stock before order arrival will not be empty, hence $x_{t,r} \leq x_t + \beta_t \leq (R + L_{ord}) \cdot D_{\max}$. The maximum container fleet size is so that we can create an ordering policy never buying any disposable. It is thus bounded by the maximum inventory position given that the number of outgoing containers is maximum. The number of outgoing containers at $r = 0$ is bounded by $z_t \leq L_{del} \cdot D_{\max}$. We hence have the upper bound $U_M := (R + L_{ord} + L_{del}) \cdot D_{\max}$ of the optimal fleet size. Any container over U_M will stay in the manufacturer stock during the whole process, hence only incurring additional purchasing and holding costs. Since we will never have more than U_M containers, we will never purchase more than U_M containers at once and the manufacturer will never have more than U_M containers in stock: $\alpha_t, y_t \leq U_M = (R + L_{ord} + L_{del}) \cdot D_{\max}$.

□

We denote by *EoH* the end of the time horizon, i.e. time $(T, 0)$. Let $\varphi^*_{t,r}(\boldsymbol{s})$ be the expected cost of an optimal policy starting from state $\boldsymbol{s}$ at time (t, r) up to EoH, and $\varphi^*_t(\boldsymbol{s}) := \varphi^*_{t,0}(\boldsymbol{s})$. Let $\varphi^*_t(\boldsymbol{s}, \boldsymbol{\gamma})$ be the minimum expected cost starting from state $\boldsymbol{s}$ and decision $\boldsymbol{\gamma}$ at time $(t, 0)$ and up to EoH, when future decisions are taken optimally. Then:

$$\varphi^*_t(\boldsymbol{s}) = \min_{\boldsymbol{\gamma}} \varphi^*_t(\boldsymbol{s}, \boldsymbol{\gamma}) \tag{7.11}$$

We denote by $\psi_{t,r}(\boldsymbol{s}, d)$ the cost incurred at time (t, r) when starting from state $\boldsymbol{s}$ and under demand d, and by $\psi_t(\boldsymbol{s}, \boldsymbol{\gamma}, d)$ the cost at time $(t, 0)$ under state $\boldsymbol{s}$, decision $\boldsymbol{\gamma}$ and demand d:

- For $r = 0$:

$$\varphi^*_t(\boldsymbol{s}, \boldsymbol{\gamma}) = \mathbb{E}_d\big[\psi_{t,r}(\boldsymbol{s}, d)\big] + \mathbb{E}_d\big[\varphi^*_{t,1}\left(\boldsymbol{s_{t,1}}[\boldsymbol{s}, \boldsymbol{\gamma}, d], \boldsymbol{\gamma}\right)\big] \tag{7.12}$$

- For $r \in [1, R-1]$:

$$\varphi^*_{t,r}(\boldsymbol{s}) = \mathbb{E}_d\big[\psi_{t,r}(\boldsymbol{s}, d)\big] + \mathbb{E}_d\big[\varphi^*_{t,r+1}\left(\boldsymbol{s_{t,r+1}}[\boldsymbol{s}, d]\right)\big] \tag{7.13}$$

The single time step cost $\psi_{t,r}$ must be defined as a function of the current state $\boldsymbol{s}$, demand d and decision $\boldsymbol{\gamma}$. The ensured manufacturer stock $Y_{t,r}$ is greater than the actual manufacturer stock as it contains the full containers which should arrive later in the current period. Therefore, the cost $\psi_{t,r}$ must already include the manufacturer holding cost of the $\Delta_{t,r}$ full containers from their planned arrival (t', r') up to the next ordering time $(t'+1, 0)$. At time $(t, 0)$, we consider in addition the holding cost for each container which has not been ordered and will thus stay idle in the manufacturer stock up to the next ordering time $(t+1, 0)$. Let $C_{man}(t, r \to R)$ denote the total manufacturer holding cost of a container from time (t, r) to time $(t+1, 0)$, so that $C_{man}(t, R-1 \to R) = C_{man}(t, R-1)$ and $C_{man}(t, R \to R) = 0$. We define the single step cost function $\psi_{t,r}$ as following:

- If $r = 0$:

$$\begin{aligned}\psi_{t,r}(\boldsymbol{s}, \boldsymbol{\gamma}, d) = (d-\delta) \cdot C_{dis}(t,0) + \delta \cdot C_{man}(t, L_{del} \to R) \\ + (x-\delta) \cdot C_{sup}(t,0) + (y+\alpha-\beta) \cdot C_{man}(t, 0 \to R)\end{aligned} \tag{7.14}$$

- If $r \in [1, R - L_{del}]$:

$$\begin{aligned}\psi_{t,r}(\boldsymbol{s}, d) = (d-\delta) \cdot C_{dis}(t,r) + \delta \cdot C_{man}(t, r + L_{del} \to R) \\ + (x-\delta) \cdot C_{sup}(t,r)\end{aligned} \tag{7.15}$$

- If $r > R - L_{del}$:

$$\begin{aligned}\psi_{t,r}(\boldsymbol{s}, d) = (d-\delta) \cdot C_{dis}(t,r) + \delta \cdot C_{man}\left(t+1, r + L_{del} - R \to R\right) \\ + (x-\delta) \cdot C_{sup}(t,r)\end{aligned} \tag{7.16}$$

Algorithm 18 computes an optimal online policy to the container purchasing problem. We note that the optimal policy corresponding to φ_t^* is computed at the same time as its expected cost, so we will equivalently say that we compute an optimal policy or its expected cost.

Theorem 7.3.2 *Under Hypotheses 2.1 [Delay] and 2.2 [Cost], Algorithm 18 computes an optimal online policy in* $O\big(T \cdot R^6 \cdot D_{\max}^6\big)$ *time.*

Proof:

By Lemma 7.3.1, the state space is $O(R^3 \cdot D_{\max}^3)$, the demand space $O(D_{\max})$ and the decision space $O(R^2 \cdot D_{\max}^2)$. There are T time steps with decision and $(R-1) \cdot T$ steps without, so the total complexity is $O\left(T \cdot R^6 \cdot D_{\max}^6\right)$. □

Algorithm 18: Optimal online Algorithm *Mdp.N*

$\varphi_T^*(\boldsymbol{s}) := 0$ for all $\boldsymbol{s}$;
for *period* t *from* $T-1$ *to* 0 **do**
  for *step* r *from* $R-1$ *to* 1 **do**
    for each *state* $\boldsymbol{s_{t,r}}$ **do**
      Compute $\varphi_{t,r}^*(\boldsymbol{s_{t,r}})$ using (7.12), (7.15) and (7.16);
  for each *state* $\boldsymbol{s_t}$ **do**
    for each *decision* $\boldsymbol{\gamma_t}$ **do**
      Compute $\varphi_t^*(\boldsymbol{s_t}, \boldsymbol{\gamma_t})$ using (7.11), (7.13) and (7.14);
    $\varphi_t^*(\boldsymbol{s_t}) := \min_{\boldsymbol{\gamma}} \left[\varphi_t^*(\boldsymbol{s_t}, \boldsymbol{\gamma})\right]$;
return $\varphi_t^*([0,0,0])$;

7.4 Convexity under Linear Cost

We consider the special case without the setup costs $C_{setup}(t)$ for container purchasing. We prove using results from the literature that the cost functions are then $L^\natural$-convex. We use an alternative and less intuitive modeling of the state, inspired from Zipkin [141]. The convexity result was already in our paper [55]. However, we discovered later a paper from Huh and Janakiraman [48] extending the results of Zipkin [141] to multi-echelon supply chains. Their state model is very close to ours, and they deduce from the $L^\natural$-convexity that the sensitivity of the optimal order policy is very small. We extend this result to our model, which allows us to further reduce the time complexity bound of our algorithms.

7.4.1 Literature Results on $L^\natural$-Convexity

We denote by $\mathcal{E}$ a one dimensional space, and by $\mathcal{E}^+$ its restriction to positive values. We denote by $\boldsymbol{e}$ the vector of '1', and by $\boldsymbol{e}_i$ the vector so that the i-th element is a '1' and every other element is a '0'.
Given two vectors $\boldsymbol{w} = [w_1, \ldots, w_n] \in \mathcal{E}^n$ and $\boldsymbol{w'} = [w'_1, \ldots, w'_n] \in \mathcal{E}^n$, we define:

$$\boldsymbol{w} \wedge \boldsymbol{w'} := \left[\min\{w_1, w'_1\}, \ldots, \min\{w_n, w'_n\}\right] \tag{7.17}$$

$$\boldsymbol{w} \vee \boldsymbol{w'} := \left[\max\{w_1, w'_1\}, \ldots, \max\{w_n, w'_n\}\right] \tag{7.18}$$

Definition 7.4.1 *A space $\mathcal{E}^n$ is* lattice *if for each $\boldsymbol{w}, \boldsymbol{w'} \in \mathcal{E}^n$ and $\lambda \in \mathcal{E}^+$, the vectors $(\boldsymbol{w} + \lambda \cdot \boldsymbol{e}) \wedge \boldsymbol{w'}$ and $\boldsymbol{w} \vee (\boldsymbol{w'} - \lambda \cdot \boldsymbol{e})$ are also in $\mathcal{E}^n$.*

Definition 7.4.2 *A function $f : \mathcal{E}^n \to \mathbb{R}$ is* $L^\natural$-convex *if $\mathcal{E}^n$ is lattice and for each $\boldsymbol{w}, \boldsymbol{w'} \in \mathcal{E}^n$ and $\lambda \in \mathcal{E}^+$, we have:*

$$f(\boldsymbol{w}) + f(\boldsymbol{w'}) \geq f\big((\boldsymbol{w} + \lambda \cdot \boldsymbol{e}) \wedge \boldsymbol{w'}\big) + f\big(\boldsymbol{w} \vee (\boldsymbol{w'} - \lambda \cdot \boldsymbol{e})\big) \tag{7.19}$$

The following theorem regroups several properties of $L^\natural$-convex functions, which can be found in the book of Simchi-Levi et al. [108] (Propositions 2.3.3, 2.3.4 and Lemma 2.3.5):

Theorem 7.4.3 ***[Simchi-Levi et al. 2013]***

1. *Any set with representation* $\{\boldsymbol{w} \in \mathcal{E}^n : \boldsymbol{l} \leq \boldsymbol{w} \leq \boldsymbol{u},\ w_i - w_j \leq v_{i,j}\ \forall i \neq j\}$ *is lattice, where* $\boldsymbol{l}, \boldsymbol{u} \in \mathcal{E}^n$ *and* $v_{i,j} \in \mathcal{E}(i \neq j)$*. In fact, any closed lattice set in the space* $\mathcal{E}^n$ *can have such a representation.*

2. *A function* $f : \mathcal{E}^n \to \mathbb{R}$ *is* $L^\natural$*-convex if and only if the function* $g(\boldsymbol{w}, \lambda) := f(\boldsymbol{w} - \lambda \cdot \boldsymbol{e})$ *is* $L^\natural$*-convex.*

3. *If* $f_1, f_2 : \mathcal{E}^n \to \mathbb{R}$ *are* $L^\natural$*-convex and* $\lambda \in \mathcal{E}^+$*, then* $\lambda \cdot f_1 + f_2$ *is also* $L^\natural$*-convex.*

4. *If* $f(\cdot, \cdot) : \mathcal{E}^n \times \mathcal{E}^m \to \mathbb{R}$ *is* $L^\natural$*-convex with respect to its first parameter, then* $g(\boldsymbol{v}) := \mathbb{E}_{\boldsymbol{w}}[f(\boldsymbol{v}, \boldsymbol{w})]$ *is also* $L^\natural$*-convex, if it is well defined.*

5. *If* $f(\cdot) : \mathcal{E}^n \to \mathbb{R}$ *is* $L^\natural$*-convex, then* $g : \mathcal{E}^{n+1} \to \mathbb{R}$ *defined by* $g(\boldsymbol{v}, \lambda) := f(\boldsymbol{v} - \lambda \cdot \boldsymbol{e})$ *is also* $L^\natural$*-convex.*

6. *If* $f(\cdot, \cdot) : \mathcal{E}^n \times \mathcal{E}^m \to \mathbb{R}$ *is* $L^\natural$*-convex, then the function* $g(\boldsymbol{v}) := \inf_{\boldsymbol{w}} f(\boldsymbol{v}, \boldsymbol{w})$ *is also* $L^\natural$*-convex.*

7. *Suppose that* $g(\boldsymbol{v}, \lambda) : \mathcal{E}^n \to \mathbb{R}$ *is* $L^\natural$*-convex, and let* $\lambda^*(\boldsymbol{v})$ *be the largest optimal solution (assuming existence) of the minimization problem* $\min_\lambda g(\boldsymbol{v}, \lambda)$*. Then, for all* $\omega \geq 0$ *and all* $i \in [1, n]$ *we have:*

$$\lambda(\boldsymbol{v}) \leq \lambda(\boldsymbol{v} + \omega \cdot \boldsymbol{e}_i) \leq \lambda(\boldsymbol{v} + \omega \cdot \boldsymbol{e}) \leq \lambda(\boldsymbol{v}) + \omega \tag{7.20}$$

Moreover, if a continuous function f is $L^\natural$-convex, then its restriction to any discrete lattice set is also $L^\natural$-convex and the properties still hold. Therefore, we can analyze the $L^\natural$-convexity on a continuous state, decision and demand space, and the results also hold for our discrete definition domains.
Chen et al. [14] prove the intuitive result that, under $L^\natural$-convex cost functions, we can use a binary search to find the optimal decision in logarithmic time with respect to the decision space.

Proposition 7.4.4 ***[Chen et al. 2014a, Lemma 8]*** *It takes* $O(\log[U - L]^k)$ *time to minimize a discrete* $L^\natural$*-convex function* $f : [L, U]^k \to \mathbb{R}^+$*.*

7.4.2 $L^\natural$-Convexity under Linear Cost

We assume that there is no setup cost and suppose first that $L_{ord} = 1$. Our state reformulation is very similar to Zipkin [141] as well as Huh and Janakiraman [48].

We set the number of full containers $\delta_{t,r} \in [0, \min\{d_{t,r}, x_{t,r}\}]$ used for demand $d_{t,r}$ as a decision variable. By Hypothesis 2.2, the value of $\delta_{t,r}$ minimizing the expected cost is:

$$\delta^*_{t,r} = \min(x_{t,r},\ d_{t,r}). \tag{7.21}$$

We denote by Y_t^α and X_t^β (resp. y_t^α and x_t^β) the variables (resp. their realizations) of the post-purchasing manufacturer inventory level and the post-ordering supplier inventory position at period t. We recall that in inventory management, the inventory level refers to the stock and the inventory position equals the stock plus the total order size. Moreover, we denote by U_t^α and u_t^α the variable and realization of the post-purchasing container fleet size:

$$\begin{aligned} Y_t^\alpha &:= Y_t + \alpha_t \\ X_t^\beta &:= X_t + \beta_t \\ U_t^\alpha &:= U_t + \alpha_t \end{aligned}$$

In order to prove the $L^\natural$-convexity of the cost functions $\varphi_{t,r}$, the state transition must be so that we add or remove a value to every parameter at the same time, in order to make use of Theorem 7.4.3.(2). In addition, the decision should be written as the minimization problem in 7.4.3.(6).
For time (t, r), we define state $\hat{\boldsymbol{s}}_{\boldsymbol{t,r}}$ as following:

- For $r = 0$:
$$\hat{\boldsymbol{s}}_{\boldsymbol{t,0}} := [x_t;\ -y_t - z_t;\ -z_t] \tag{7.22}$$
- After purchasing:
$$\hat{\boldsymbol{s}}^{\boldsymbol{\alpha}}_{\boldsymbol{t,0}} := [x_t,;\ -y_t - z_t - \alpha_t,;\ -z_t] \tag{7.23}$$
- After ordering:
$$\hat{\boldsymbol{s}}^{\boldsymbol{\beta}}_{\boldsymbol{t,0}} := [x_t - \beta_t,;\ -y_t - z_t - \alpha_t - \beta_t,;\ -\beta_t] \tag{7.24}$$
- For $r = L_{ord}$:
$$\hat{\boldsymbol{s}}_{\boldsymbol{t,L_{ord}}} := \left[\hat{x}^\beta_{t,0} + \hat{z}^\beta_{t,0};\ \hat{y}^\beta_{t,0};\ 0\right] - \delta^*_{t,L_{ord}-1} \cdot \boldsymbol{e} \tag{7.25}$$
- For $r \in [L_{ord} + 1,\ R - L_{del}]$:
$$\hat{\boldsymbol{s}}_{\boldsymbol{t,r}} := [\hat{x}_{t,r-1};\ \hat{y}_{t,r-1};\ 0] - \delta^*_{t,r-1} \cdot \boldsymbol{e} \tag{7.26}$$
- For $r \in [R - L_{del} + 1,\ R]$:
$$\hat{\boldsymbol{s}}_{\boldsymbol{t,r}} := [\hat{x}_{t,r-1};\ \hat{y}_{t,r-1};\ \hat{z}_{t,r-1}] - \delta^*_{t,r-1} \cdot \boldsymbol{e} \tag{7.27}$$

After ordering, we have $\hat{x}^{\beta}_{t,0} = x_t + \beta_t$ and $\hat{z}^{\beta}_{t,0} = \beta_t$. Given demand realization $d_{t,0}$, then maximum of $\delta^{var}_{t,r}$ equals the minimum of $d_{t,r}$ and $\hat{z}_{t,r} = \hat{x}_{t,r} - \hat{z}_{t,r}$. For $t \in [0, T[$, our new decisions ares:

$$\hat{\gamma}_t := [\hat{\alpha}_t, \hat{\beta}_t] \tag{7.28}$$

$$\hat{\alpha}_t := -\alpha_t - x_t - y_t - z_t \tag{7.29}$$

$$\hat{\beta}_t := -\beta_t \tag{7.30}$$

We define $\hat{\varphi}^*_{t,r}$, $\hat{\varphi}^*_t$ and $\hat{\psi}_{t,r}$ relatively to the updated states $\hat{\boldsymbol{s}}_{\boldsymbol{t,r}}$ similarly to $\varphi^*_{t,r}$, φ^*_t and $\psi_{t,r}$. By definition, we have then:

$$\forall t, x, y, z \ \varphi^*_t(x, y, z) = \hat{\varphi}^*_t(x, -y - z, -z) \tag{7.31}$$

Lemma 7.4.5 *The definition domain of state $\hat{\boldsymbol{s}}_{\boldsymbol{t,r}}$ is lattice for every $t \in [0, T[$ and $r \in [0, R[$.*

Proof:

Consider $t \in [0, T[$. The definition domain of $\hat{\boldsymbol{s}}_{\boldsymbol{t,r}}$ for all $r \in [0, R[$ is as following, by Lemma 7.3.1:

- For $\hat{x}_{t,r}$:
$$0 \leq \hat{x}_{t,r} \leq (R + L_{ord}) \cdot D_{\max}$$
- For $\hat{y}_{t,r}$:
$$\big(2 \cdot (R + L_{del}) + L_{ord}\big) \cdot D_{\max} \leq \hat{y}_{t,0} \leq 0$$
- For $\hat{z}_{t,r}$:

$$\begin{aligned} r = 0 : \hat{y}_{t,0} \leq \hat{z}_{t,0} \leq 0 \\ \hat{y}^{\alpha}_{t,0} \leq \hat{z}^{\alpha}_{t,0} \leq 0 \\ 0 \leq \hat{z}^{\beta}_{t,0} \leq \hat{x}^{\beta}_{t,0} \\ r \in [L_{ord}, R - L_{del}] : \hat{y}_{t,r} \leq \hat{z}_{t,r} (= 0) \leq 0 \\ r \in]R - L_{del}, R - 1[: \hat{y}_{t,r} \leq \hat{z}_{t,r} \leq 0 \end{aligned}$$

  Note that we need to extend the definition space of $\hat{z}_{t,r}$, $r \in [L_{ord}, R - L_{del}]$ to negative values so that the state $[\hat{x}_{t,r-1}, \hat{y}_{t,r-1}, 0] - \delta_{t,r-1}$ belongs to the domain of $\hat{\boldsymbol{s}}_{\boldsymbol{t,r}}$

The definition domains are as in Theorem 7.4.3.(1) and are hence lattice.

□

Proposition 7.4.6 *Under Hypotheses 2.1 [Delay] and 2.2 [Cost], the functions $\hat{\varphi}^*_{t,r}$, $\hat{\varphi}^*_t$ and $\hat{\psi}_{t,r}$ are $L^{\natural}$-convex for every $t \in [0, T[$, $r \in [0, R[$.*

Proof:

By Lemma 7.4.5, the definition domains are lattice. Following Equations (7.14) to (7.16), the single time step cost function $\hat{\psi}_{t,r}$ for state $\hat{\boldsymbol{s}}_{\boldsymbol{t,r}}$ is the solution of a minimization problem over $\delta^{var}_{t,r}$. Since the minimization problem is a linear combination of the parameters, it is $L^\natural$-convex by Theorem 7.4.3.(6).
From the post-decision state at time (t,r) to the next state at time $(t,r+1)$, the state is updated by removing $\delta^{var}_{t,r}$ to all state parameters. Therefore, for $r \in [1,R[$:

$$\hat{\varphi}_{t,r}(\hat{\boldsymbol{s}}_{\boldsymbol{t,r}}) = \mathbb{E}_{d_{t,r}}\big[\hat{\psi}_{t,r}(\hat{\boldsymbol{s}}_{\boldsymbol{t,r}}, d_{t,r})\big] + \mathbb{E}_{d_{t,r}}\big[\inf_{\delta^{var}_{t,r}} \hat{\varphi}_{t,r+1}(\hat{\boldsymbol{s}}_{\boldsymbol{t,r}} - \delta^{var}_{t,r})\big] \tag{7.32}$$

By Theorem 7.4.3.(2,3,4,6), we have that if $\hat{\varphi}_{t+1,0}$ is $L^\natural$-convex then the post-decision function $\hat{\varphi}^{\beta}_{t,0}$ at time $(t,0)$ is also $L^\natural$-convex.
Likewise, the post-purchasing cost function $\hat{\varphi}^{\alpha}_{t,0}$ is $L^\natural$-convex by Theorem 7.4.3.(2,6), as it can be written:

$$\hat{\varphi}_{t,0}(\hat{\boldsymbol{s}}_{\boldsymbol{t,0}}{}^{\alpha})^{\alpha} = \inf_{\hat{\beta}_t} \hat{\varphi}^{\beta}_{t,0}(\hat{\boldsymbol{s}}_{\boldsymbol{t,0}}{}^{\alpha} - \hat{\beta}_t) \tag{7.33}$$

Thus, the pre-purchasing cost function $\hat{\varphi}_{t,0}$ is $L^\natural$-convex by Theorem 7.4.3.(2), as it is a minimization problem over a single variable.

□

We deduce that the problem is $L^\natural$-convex. Consequently, in Algorithm 18 we can use a binary search to find the optimal decisions. Therefore, without setup cost, we can compute an optimal online policy in $O\big(T \cdot R^4 \cdot D^4_{\max} \cdot \log[D_{\max} \cdot R]^2\big)$ time.
When $L_{ord} > 1$, the difficulty lies within the preliminaries time steps $r \in [1, L_{ord}[$. We need the post-ordering state to contain both values x_t and $x_t + \beta_t$ to be able to compute the maximum value of $\delta^{var}_{t,r}$. This can be done by replacing the decision $\hat{\beta}_t = -\beta_t$ by a two dimensional decision, whose parameters correspond to the leftover manufacturer stock and the new supplier inventory position.

Theorem 7.4.7 *Without setup costs, Algorithm 18 computes an optimal online policy in $O\big(T \cdot R^4 \cdot D^4_{\max} \cdot \log[D_{\max} \cdot R]^2\big)$ time.*

Furthermore, we deduce from the $L^\natural$-convexity that the partial derivative of the optimal purchasing $\alpha^*_t = \alpha^*_t(\hat{x}, \hat{y}, \hat{z})$ and order size β^*_t is in $\{-1, 0\}$ with respect to the supplier stock x_t and in $\{0, 1\}$ with respect to the manufacturer stock y_t and the outgoing fleet z_t of full containers:

Corollary 7.4.8 *Under Hypotheses 2.1 [Delay], 2.2 [Cost] and without setup cost, the optimal decisions α^* and β^* are so that:*

$$0 \leq \frac{\partial}{\partial \hat{z}}\beta^* \leq \frac{\partial}{\partial \hat{y}}\beta^* \leq -\frac{\partial}{\partial \hat{x}}\beta^* \leq 1 \tag{7.34}$$

$$0 \leq \frac{\partial}{\partial \hat{z}}\alpha^* \leq \frac{\partial}{\partial \hat{y}}\alpha^* \leq -\frac{\partial}{\partial \hat{x}}\alpha^* \leq 1 \tag{7.35}$$

Proof:

The result follows from the $L^\natural$-convexity from the cost functions $\hat{\varphi}_t$.

□

We conclude:

Theorem 7.4.9 *Under Hypotheses 2.1 [Delay], 2.2 [Cost] and without setup cost, We can compute an optimal online policy in $O\big(T \cdot R^4 \cdot D_{\max}^4\big)$ time.*

Proof:

By corollary 7.4.8, the gradient of the purchasing and ordering decisions is between -1 and 1 for every parameter. Therefore, after computing the optimal purchase and order sizes for a state, the optimal decisions in the neighborhood can be computed by considering a constant number of possibilities. Therefore, the amortized computation time is $O\big(T \cdot R^4 \cdot D_{\max}^4\big)$.

□

We call *incremental search* the search we use to compute the order size in Theorem 7.4.9, as opposed to the binary search.

Remark 7.4.10 *The incremental search from Theorem 7.4.9 is more efficient than the binary search used for Theorem 7.4.7. However, the binary search computes the optimal decision for every state separately, which can be used with approimation methods proposed in Chapter 8.*
In the following section, we describe an algorithm whose cost functions are $L^\natural$-convex, and uses a binary search to compute the optimal order size, as we have done in our paper [55]. The complexity bounds using the incremental search is obtained by removing the logarithmic terms.

7.5 Offline Heuristic

In this section, we present two algorithms for the offline strategy. The first algorithm is optimal, but is not applicable in practice due to an exponential running time. The second policy is our heuristic to compute an efficient purchasing plan. We propose two algorithms computing the same offline policy, with concurring time complexities. All algorithms are based on the Markov decision processes described for the online algorithm.

7.5.1 Optimal Algorithm

We compute an optimal offline policy using an exhaustive seach over every posible purchasing plan. For each purchasing plan, the purchasing cost is fixed, so the cost functions are $L^\natural$-convex and the optimal ordering decisions can be found using a binary search (or an incremental search) as described in the previous section. Moreover the container fleet size is known and we only need two parameters $X_{t,r}$ and $Z_{t,r}$ per state. The state space is $O(R^2 \cdot D_{\max}^2)$. By theorem 7.4.7, it takes $O\big(T \cdot R^3 \cdot D_{\max}^3 \cdot \log[D_{\max} \cdot R]\big)$ time to compute the expected cost and the optimal ordering policy under a fixed container purchasing plan. The backward programming takes $O(T \cdot R \cdot D_{\max})$ time per state and decision, and the ordering decision takes $O(\log[D_{\max} \cdot R])$ time. We denote by $Opt.F$ this optimal offline algorithm. Since the complexity of $Opt.F$ is exponential, we can only run it for very small test instances.

7.5.2 Heuristic Algorithm

We first describe the heuristic using the same states $\boldsymbol{s_{t,r}}$ as in Algorithm 18. Afterward, we show the $L^\natural$-convexity of the cost functions using the states $\hat{\boldsymbol{s}}_{\boldsymbol{t,r}}$. Recall that computing a policy is equivalent to computing its expected cost so in our algorithm we look for the expected cost of the optimal policies. We denote by $\varphi^*[k_2](\boldsymbol{s_{k_2}})$ the cost of the optimal offline policies from state $\boldsymbol{s_{k_2}}$ at period k_2 to EoH. The policy $\varphi^*[k_2]$ is so that the container fleet size after purchasing at period k_2 is the same for each starting state $\boldsymbol{s_{k_2}}$ at period k_2. An offline policy defined over periods $[k_2, T[$ gives the purchasing plan starting from period k_2. In particular, it raises the container fleet size to $u^\alpha(k_2)$ for each state at period k_2 with initial fleet size below $u^\alpha(k_2)$. The policy supposes hence that the container fleet size before placement k_2 is at most $u^\alpha(k_2)$.

We denote by $\varphi^*[k_1, k_2, u^\alpha](\boldsymbol{s_{k_1}})$ the expected cost of an optimal offline policy from state $\boldsymbol{s_{k_1}}$ at period k_1 to EoH so that k_1 and k_2 are the first two placements and so that the container fleet size after purchasing at period k_1 is u^α. We can compute $\varphi^*[k_1, k_2, u^\alpha]$ from $\varphi^*[k_2]$ using backward dynamic programming.

Suppose that we know the cost function $\varphi^*[k_2]$. For a fixed k_1, there are two values $k_2^*(k_1) > k_1$ and $u^{\alpha,*}(k_1)$ inducing an optimal offline policy with cost $\varphi^*[k_1] := \varphi^*\left[k_1, k_2^*(k_1), u^{\alpha,*}(k_1)\right]$ from period k_1 to EoH. The optimal policy starting from placement k_1 and with next placement $k_2^*(k_1)$ raises the container fleet size of the process at period k_1 up to $u^{\alpha,*}(k_1) < u^\alpha(k_2^*(k_1))$. As we may expect, different initial states $\boldsymbol{S_{k_1}}$ at period k_1 may lead to different best values of $k_2(k_1)$ and $u^\alpha(k_1)$. When using a Markov decision process, we hence need the probability distribution of $\boldsymbol{S_{k_1}}$ at period k_1 to find $k_2^*(k_1)$. Contrary to Wagner and Within [128] as well as Vargas [125], the optimal solution after purchasing period k_1 depends on the optimal solution

before purchasing time k_1. The state probability distribution at period k_1 can only be derived from an optimal policy up to period k_1. But, in order to determine it, we need the expected cost $\varphi[k_1](\boldsymbol{s_{k_1}})$ for each state $\boldsymbol{s_{k_1}}$ at period k_1, which is what we actually want to compute.
Our heuristic neglects the influence of the state $\boldsymbol{S_{k_1}}$ on the optimal container fleet size after period k_1 and on the next purchasing timing. Hence, we approximate the optimal container fleet size $u^{\alpha,*}(k_1)$ after purchasing and the next purchasing time $k_2^*(k_1)$ as the cost minimizing values $u^{\alpha,ref}(k_1)$ and $k_2'(k_1)$ when starting from the *reference state* $\boldsymbol{s'_{k_1}} := [0,0,0]$ at time $(k_1, 0)$. We would like $u^{\alpha,ref}(k_1)$ and $k_2'(k_1)$ to be close to $u^{\alpha,*}(k_1)$ and $k_2^*(k_1)$. Then, for each other state $\boldsymbol{s_{k_1}}$, we set $u_{k_1} := x_{k_1} + y_{k_1} + z_{k_1}$, $\alpha_{k_1}(\boldsymbol{s_{k_1}}) := u^{\alpha,ref}(k_1) - u_{k_1}$ and compute the optimal order size β_{k_1} given $u^{\alpha,ref}(k_1)$ and $k_2'(k_1)$. Since we are only computing a suboptimal solution, we use the notation φ instead of φ^*. Our quasi-offline policy computes $\varphi[k_1, k_2]$ from $\varphi[k_2]$, and deduce $k_2'(k_1)$ and $u^{\alpha,ref}(k_1)$. The optimal solution is approximated by:

$$\varphi[k_1] := \varphi\left[k_1, k_2'(k_1), u^{\alpha,ref}(k_1)\right] \tag{7.36}$$

Our algorithm is initialized with the EoH costs:

$$\forall \boldsymbol{s}: \ \varphi[T](\boldsymbol{s}) = 0 \tag{7.37}$$

Without loss of generality, we can assume that some containers are purchased at period 0. To relax this assumption, we add a dummy period -1 with zero demand, zero manufacturer holding cost and infinite supplier holding cost. Our offline heuristic policy has then expected cost $\varphi[0]([0,0,0])$. We now explain how to efficiently compute $\varphi[k_1, k_2]$ from $\varphi[k_2]$ using backward programming and the $L^\natural$-convexity. For $k_2 \in [1, T]$ and given cost $\varphi[k_2]$, we define $\phi'[k_1, k_2](\boldsymbol{s})$ as the expected cost of the optimal policy starting from state $\boldsymbol{s}$ at time $(k_1, 1)$ up to EoH without purchasing containers before period k_2 and so that the expected cost from state $\boldsymbol{s_{k_2}}$ at period k_2 up to EoH is given by $\varphi[k_2](\boldsymbol{s_{k_2}})$.

Lemma 7.5.1 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Consider a period k_2 and suppose that the costs $\varphi[k_2]$ are known and $L^\natural$-convex. Then, the costs $\phi'[k_1, k_2]$ for all $k_1 \in [0, k_2 - 1]$ are also $L^\natural$-convex and can be computed altogether in $O\big(T \cdot R^4 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]\big)$ time ($O\big(T \cdot R^4 \cdot D_{\max}^4\big)$ for the incremental search).*

Proof:

For a fixed k_2, all $\phi'[k_1, k_2]$ can be computed altogether with a single backward dynamic program starting from $\varphi[k_2]$ up to time $(k_1, 1)$, with $O(R \cdot T)$ iterations. Since we are not purchasing any container while computing $\phi'[k_1, k_2]$, we prove using the same reasoning and the same states $\hat{s}_{t,r}$ as

in Sect. 4, that the cost of all $\phi'[k_1, k_2]$ are $L^\natural$-convex. By Lemma 7.3.1, the state space is $O(R^3 \cdot D_{\max}^3)$ and the decision space is $O(R \cdot D_{\max})$. Due to the $L^\natural$-convexity, we get the best order size in $O\left(\log[R \cdot D_{\max}]\right)$ time. The total time complexity is then $O\big(T \cdot R^4 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]\big)$.

□

Lemma 7.5.2 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Consider periods k_1 and k_2 and suppose that the costs $\phi'(k_1, k_2)$ are known and $L^\natural$-convex. Then the costs $\varphi[k_1]$ are also $L^\natural$-convex and can be computed in $O\big(T \cdot R^3 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]\big)$ time ($O\big(T \cdot R^3 \cdot D_{\max}^4\big)$ for the incremental search).*

Proof:

The policy relative to $\varphi[k_1]$ already includes the setup cost $C_{setup}(k_1)$, thus the computation of the costs $\varphi[k_1]$ only consider linear costs and the $L^\natural$-convex cost functions $\phi'(k_1, k_2)$. Using the same reasoning as in Lemma 7.5.1, the cost $\varphi[k_1, k_2, u^\alpha](\hat{\boldsymbol{s}})$ is $L^\natural$-convex for each state $\hat{\boldsymbol{s}}$ and every k_1, k_2, u^α. We compute $k_2'(k_1)$ and $u^{\alpha,ref}(k_1)$ using a binary search on the purchasing and ordering decisions over a single time step and a single state, which takes $O(T \cdot D_{\max} \cdot \log[R \cdot D_{\max}]^2)$ time.
We then compute $\varphi[k_1] := \varphi[k_1, k_2'(k_1), u^{\alpha,ref}(k_1)]$ for $O(R^3 \cdot D_{\max}^3)$ states and with a binary search for the order size, which takes $O(R^3 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}])$ time. The total complexity is $O\big(T \cdot R^3 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]\big)$.

□

Proposition 7.5.3 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Every cost $\phi'(k_1, k_2)$ for $0 \leq k_1 < k_2 \leq T$ is $L^\natural$-convex and can be computed altogether in $O\left(T^2 \cdot R^3 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]\right)$ time ($O\left(T^2 \cdot R^3 \cdot D_{\max}^4\right)$ for the incremental search).*

Proof:

This proposition is proven by recursion. For $k_2 = T$, the costs $\varphi[T] = 0$ are $L^\natural$-convex. Therefore the costs $\phi'[k_1, T]$ for $k_1 \in [0, T-1]$ are $L^\natural$-convex and we compute them them $O\left(T \cdot R^4 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]\right)$ time. Consider now a value of $k_2 \in [1, T-1]$ and suppose that for all $k_4 \in [k_2 + 1, T]$ and all $k_3 \in [0, k_2]$, the cost $\phi'(k_3, k_4)$ is known and $L^\natural$-convex. In particular, the cost $\phi'\big(k_2, k'(k_2)\big)$ is $L^\natural$-convex, and by Lemma 7.5.2 $\varphi(k_2)$ is $L^\natural$-convex and computed in $O\big(T \cdot R^4 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]\big)$ time. By Lemma 7.5.1, for all $k_1 \in [0, k_2 - 1]$, the costs $\phi'(k_1, k_2)$ are also $L^\natural$-convex and take $O\left(T \cdot R^4 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]\right)$ time to compute. We conclude that all costs $\phi'(k_1, k_2)$ for $0 \leq k_1 < k_2 \leq T$ can be computed altogether in $O\left(T^2 \cdot R^4 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]\right)$ time.

□

Our offline heuristic policy is computed by Algorithm 19.

Algorithm 19: offline Algorithm *Mdp.F*

$\varphi^*_T(\boldsymbol{s}) := 0$ for all $\boldsymbol{s}$;
for *period* k *from* $T-1$ *to* 0 **do**
  Compute $\phi'[k, k+1]$ from $\varphi[k+1]$;
  for *period* t *from* $k-1$ *to* 0 **do**
    Compute $\phi'(t, k+1)$ from $\phi'(t+1, k+1)$;
  Compute $k'_2(k)$ and $u^{\alpha,ref}(k)$ from $\{\phi'[k, k_2],\ k_2 > k\}$;
  Compute $\varphi(k)$ from $\phi'[k, k'_2(k)]$ and $u^{\alpha,ref}(k)$;
return $\varphi(0)[0, 0, 0]$;

Theorem 7.5.4 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Algorithm Mdp.F computes an offline policy with the time complexity* $O\big(T^2 \cdot R^4 \cdot D^4_{\max} \cdot \log[R \cdot D_{\max}]\big)$. *(*$O\big(T^2 \cdot R^4 \cdot D^4_{\max}\big)$ *for the incremental search).*

Proof:

This theorem follows directly from Proposition 7.5.3 and Lemma 7.5.2.

□

Remark 7.5.5 *If for every* $t \in [0, T[$, *all states* $\hat{\boldsymbol{s}}_{\boldsymbol{t}}$ *induce the same purchase size and next placement time, then Algorithm 19 computes an optimal policy.*

7.5.3 Alternative Heuristic

We present an alternative algorithm to compute the same offline policy. It uses the fact that we do not need the $O(R^3 \cdot D^3_{\max})$ values of φ_{k_2} to compute $\phi'[k_1, k_2]$. We only need $O(R^2 \cdot D^2_{\max})$ values $\varphi_{u^\alpha(k_2)}[k_2]$ corresponding to the value of the state after raising the container fleet size to $u^\alpha(k_2)$ containers. Therefore, we can remove from the state the parameter $\hat{Z}_{k_2} := -Y_{k_2} - Z_{k_2}$. with the biggest definition domain, and store the container fleet size after placement $u^\alpha(k_2)$ instead.
The price of this state space reduction is that, when computing the optimal purchasing for the reference state, we have to recompute the value $\phi'_{u^\alpha}[k_1, k_2]$ for each considered purchase size, hence adding a factor $O(\log[R \cdot D_{\max}])$ back to the complexity. We note that now $\phi'_{u^\alpha}[k_1, k_2]$ contains the purchase size at period k_2.
Moreover, for a fixed value k_2, we do not consider the same fleet sizes for different values of k_1. Therefore, we need to compute $\phi'_{u^\alpha}[k_1, k_2]$ separately for every value $k_1 \in [0, k_2[$. We recall that in Algorithm $Mdp.F$, we generate every value $\varphi'[k_1, k_2]$ while computing $\varphi'[0, k_2]$.
We define our new end of horizon costs as:

$$\forall u^\alpha \in [0, U_M], \forall x, y, z ::\ \phi_{T,u^\alpha}[\hat{x}, \hat{y}] = 0 \tag{7.38}$$

Since this alternative only consists in removing one parameter from the state space in exchange for computing a backward dynamic program several times, we deduce:

Lemma 7.5.6 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Consider* $k_2 \in [1, T]$ *and* $u^\alpha \in [0, U_M]$. *If the function* $\varphi_{u^\alpha}[k_2]$ *is* $L^\natural$*-convex, then* $\varphi'_u[k_1, k_2]$ *is also* $L^\natural$*-convex and can be computed in* $O(T \cdot R^3 \cdot D^3_{\max} \cdot \log[R \cdot D_{\max}])$ *time, for all* $k_1 \in [0, k_2[$ *and* $u \in [0, u^\alpha]$.

Lemma 7.5.7 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Consider period* k_1 *and suppose that the cost functions* $\phi'_{u^\alpha}[k_1, k_2]$ *are* $L^\natural$*-convex for all* $k_2 \in]k_1, T]$. *Then the cost function* $\varphi_{u^\alpha}[k_1]$ *is also* $L^\natural$*-convex and can be computed in* $O(R^2 \cdot D^3_{\max} \cdot \log[R \cdot D_{\max}])$ *time.*

Algorithm 20: Algorithm $Mdp.F^{bis}$, computing an offline policy

```
φ_{u^α}[T](s) := 0 for all s, u^α;
u^α(T) := U_M;
for period k_1 from T − 1 to 0 do
    for period k_2 from k_1 to T − 1 do
        for fleet u^α in [0, u^α(k_2)] as a binary search do
            Compute φ'_{u^α}[k_1, k_2] from φ_{u^α(k_2)}[k_2];
            Compute φ_{u^α}[k_1, k_2] from φ'_{u^α}[k_1, k_2];
            Stop if φ_{u^α}[k_1, k_2] minimizes the costs over u^α;
        u^α(k_1, k_2) := u^α;
    k_1(k_2) := arg min [φ_{u^α(k_1,k_2)}[k_1, k_2] : k_2 ∈ [k_1, T − 1]];
    u^α(k_1) := u^α(k_1, k_2(k_1)) Compute φ_{u^α(k_1)}[k_1] from
     φ_{u^α(k_1,k_2)}[k_1, k_2];
Compute φ[0]([0, 0, 0]) from φ_{u^α(0)}[0];
return φ[0]([0, 0, 0]);
```

Theorem 7.5.8 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Algorithm* $Mdp.F^{bis}$ *computes the same policy as Algorithm* $Mdp.F$, *but in* $O(T^3 \cdot R^3 \cdot D^3_{\max} \cdot \log[R \cdot D_{\max}]^2)$ *time.*

Remark 7.5.9 *Compared to Algorithm* $Mdp.F$, *Algorithm* $Mdp.F^{bis}$ *multiplies the running time by factor* $O\big(T \cdot \log[R \cdot D_{\max}]/(R \cdot D)\big)$, *where the* $O(R \cdot D)$ *results from removing a state parameter. Which one of these algorithm is faster depends on the size of the demands and time horizon.*

7.6 Hybrid Algorithms

We present here two quasi-offline algorithms, a slow optimal algorithm and a faster heuristic. Both are very similar to their offline counterpart. To avoid

repetition, we refer to the offline strategy for the definitions and the use of the functions φ'. Afterward, we briefly describe an optimal quasi-online algorithm using the same structure as the quasi-offline heuristic algorithm.

7.6.1 Optimal Quasi-Offline Algorithm

The optimal quasi-offline algorithm computes an optimal quasi-offline policy by iteratively considering every possible set of purchasing times. For each set of purchasing times, the purchasing costs are fixed, and we can compute the optimal purchase sizes and the optimal order policy in $O\big(T \cdot R^3 \cdot D_{\max}^4 \cdot \log^2[D_{\max} \cdot R]\big)$ time, when using a binary search for the purchase and order sizes. Since there are T periods, there are 2^T possible sets of purchasing times. Thus, there are far fewer possible sets of possibilities and each possibility takes $O(D_{\max} \cdot \log[D_{\max} \cdot R])$ more time. This algorithm should be significantly faster than our optimal offline policy but still not fast enough. In our simulations, we denote this optimal quasi-offline algorithm by $Opt.Qf$.

7.6.2 Quasi-Offline Heuristic

We denote here by $\varphi^*[k_1, k_2][\boldsymbol{s_{k_1}}]$ the expected cost of an optimal quasi-offline policy from state $\boldsymbol{s_{k_1}}$ at period k_1 to EoH so that k_1 and k_2 are the first two placements. We denote by $\varphi^*[k_2](\boldsymbol{s_{k_2}})$ the optimal quasi-offline policy from state $\boldsymbol{s_{k_2}}$ at period k_2 to EoH. Given the costs $\varphi^*[k_2]$, we compute $\varphi^*[k_1, k_2]$ using backward dynamic programming. For a fixed value k_1 there is a value $k_2^*(k_1) > k_1$ inducing an optimal policy with cost $\varphi^*[k_1] := \varphi^*\left[k_1, k_2^*(k_1)\right]$ from period k_1 to EoH. Like for the offline strategy, different values of $\boldsymbol{S_{k_1}}$ at period k_1 may lead to different best values of $k_2(k_1)$. When using a Markov decision process, we thus need the probability distribution of $\boldsymbol{S_{k_1}}$ at period k_1 to find $k_2^*(k_1)$.
We use the same reference state $\boldsymbol{s'_{k_1}} := [0, 0, 0]$ to choose the next placement $k_2'(k_1)$ after k_1. We then set:

$$\varphi^*[k_1] := \varphi^*[k_1, k_2'(k_1)]$$

and compute the optimal order size β_{k_1} for each state at period k_1. For the quasi-offline heuristic, we then also use the notation φ instead of φ^*. Our algorithm is initialized with the EoH costs:

$$\forall \boldsymbol{s}: \ \varphi[T](\boldsymbol{s}) = 0 \tag{7.39}$$

Without loss of generality, we can assume that some containers are purchased at period 0. Indeed, we can relax this assumption by adding a dummy period -1 with zero demand, zero manufacturer holding cost and infinite supplier holding costs. Our quasi-offline heuristic policy has then expected cost $\varphi[0]\,([0, 0, 0])$.

We define the functions $\varphi'[k1, k2]$ as the optimal policy from period k_1 to EoH given a fixed policy $\varphi[k_2]$ starting from period k_2 so that k_1 and k_2 are the first two placements.

Lemma 7.6.1 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Consider a period k_2 and suppose that the costs $\varphi[k_2]$ are known and $L^\natural$-convex. Then, the costs $\phi'[k_1, k_2]$ for all $k_1 \in [0, k_2 - 1]$ are also $L^\natural$-convex and can be computed altogether in $O\big(T \cdot R^4 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]\big)$ time ($O\big(T \cdot R^4 \cdot D_{\max}^4\big)$ for the incremental search).*

Proof:

Same as Lemma 7.5.1.

□

Lemma 7.6.2 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Consider periods k_1 and k_2 and suppose that the cost functions $\phi'(k_1, k_2)$ are known and $L^\natural$-convex. Then the costs $\varphi[k_1]$ are also $L^\natural$-convex and can be computed in $O\big(T \cdot R^3 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]^2\big)$ time ($O\big(T \cdot R^3 \cdot D_{\max}^4\big)$ for the incremental search).*

Proof:

Same as Lemma 7.5.2.

□

Proposition 7.6.3 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Every cost $\phi'(k_1, k_2)$ for $0 \leq k_1 < k_2 \leq T$ is $L^\natural$-convex and can be computed altogether in $O\left(T^2 \cdot R^3 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]^2\right)$ time ($O\left(T^2 \cdot R^3 \cdot D_{\max}^4\right)$ for the incremental search).*

Proof:

Same as Lemma 7.5.2.

□

Our quasi-offline heuristic policy is computed by Algorithm 21.

Algorithm 21: Quasi-offline Algorithm *Mdp.Qf*

```
φ*_T(s) := 0 for all s;
for period k from T − 1 to 0 do
    Compute φ'[k, k + 1] from φ[k + 1];
    for period t from k − 1 to 0 do
        Compute φ'(t, k + 1) from φ'(t + 1, k + 1);
    Compute k'_2(k) from {φ'[k, k_2], k_2 > k};
    Compute φ(k) from φ'[k, k'_2(k)];
return φ(0)[0, 0, 0];
```

Theorem 7.6.4 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Algorithm Mdp.Qf computes a quasi-offline policy with the time complexity* $O\big(T^2 \cdot R^4 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]^2\big)$ *(*$O\big(T^2 \cdot R^4 \cdot D_{\max}^4\big)$ *for the incremental search).*

Remark 7.6.5 *If for every* $t \in [0, T[$*, all state* $\hat{\boldsymbol{s}}_{\boldsymbol{t}}$ *induce the same next placement time, then Algorithm 21 computes an optimal policy. this is the case in most application scenario, due to the weak impact of the state at placement to the system cost after one period.*

Remark 7.6.6 *When we use a binary search, the quasi-offline algorithm has a higher complexity bound than the offline algorithm, because the offline heuristic doe not need to compute the optimal purchase size for every state.*

7.6.3 Quasi-Online Algorithm

Computing an optimal quasi-offline policy follows the same pattern as Algorithm $Mdp.Qf$. We recall that the quasi-offline Algorithm approximate the best next placement as the best next placement $k_2 := k_2'(k_1)$ for a reference state. Then, we approximate the cost $\varphi[k_1]$ assuming that k_2 is the placement following k_1. In the case of a quasi-online policy, we do not need k_2 to be fixed, so the optimal policy can be fixed by letting the next placement k_2 depend on the current placement k_1. It follows that Algorithm $Mdp.Qn$ computes an optimal quasi-online policy. The $O(T^2)$ factor in the quasi-offline algorithm comes from the computation of the function ϕ', which is computed the same way for both hybrid algorithms. We deduce that the complexity bound of the two algorithms is the same, despite the quasi-online algorithm being expected to be slower in practice.

Algorithm 22: Quasi-online Algorithm $Mdp.Qn$

```
φ*_T(s) := 0 for all s;
for period k from T − 1 to 0 do
    Compute φ'[k, k+1] from φ*[k+1];
    for period t from k − 1 to 0 do
        Compute φ'(t, k+1) from φ'(t+1, k+1);
    foreach state s do
        Compute k*_2(k, s) from {φ'[k, k_2], k_2 > k};
        Compute φ*(k)[s] from φ'[k, k*_2(k)];
return φ*(0)[0, 0, 0];
```

Theorem 7.6.7 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Algorithm Mdp.Qn computes an optimal quasi-online policy with the time complexity* $O\big(T^2 \cdot R^4 \cdot D_{\max}^4 \cdot \log[R \cdot D_{\max}]^2\big)$ *(*$O\big(T^2 \cdot R^4 \cdot D_{\max}^4\big)$ *for the incremental search).*

Remark 7.6.8 *The online algorithm has a space complexity of* $O\big(T \cdot R^3 \cdot D^3_{\max}\big)$*, since there is a different decision to make for each state of each period. For the non-online policies, a naive approach requires a space* $O\big(T^2 \cdot R^3 \cdot D^3_{\max}\big)$*, since there is one decision for each state of each period and given each future decision time. Nevertheless, for an offline and quasi-offline policy, we can reduce the memory space to* $O\big(T \cdot R^3 \cdot D^3_{\max}\big)$ *by firstly computing the optimal purchasing times and quantities from the reference state, and then recomputing the order size given the future decision times.*
This is not possible for a quasi-online policy, where every order size not only depends on the state and the period, but also on the future purchasing time. For instance, if period k *is a placement, then an optimal quasi-online policy may order at period* $k-1$ *every containers in the manufacturer stock. In contrast, if the next placement is later, the policy will save some containers and order them at period* k*.*

7.7 Simulations

In this section, we simulate and compare the efficiency of our algorithms. Our first objective is to compare our offline and quasi-offline heuristics with the slower optimal algorithms. We also compare how our offline heuristic performs compared to a simple offline algorithm using no disposable. Our second objective is to compare the performance of the four strategies to each other. In particular, in a traditional lot-sizing problem, the quasi-offline strategy has significantly lower cost than the offline policy because it allows to adapt the inventory level to the past demands. Instead, in our container management problem the containers used for a demand are still in the system so the quasi-offline policy may not be as meaningful as in a traditional lot-sizing problem. Our objective here is to quantify the influence of delaying the decision of the purchase quantity to the system cost.

7.7.1 Data

We denote by Algorithm $Ndis.F$ the best offline policy avoiding disposables at all costs. As a consequence, the order size follows immediately from the supplier stock. This simple algorithm is meaningful in our simulations where the support of each demand variable is a small interval. We note that this algorithm can be adapted to larger supports by approximating the largest demand value by a quantile. We use an algorithm similar to the Wagner-Within algorithm [128] to deduce the purchasing plan from these minimum fleets, in $O(T^2)$ time. A forward dynamic program computes the expected cost of the policy. Since the fleet size is known for every period, we only need two state parameters so the complexity is $O(T \cdot R^3 \cdot D^3_{\max})$.
We use a virtual data set in order to highlight the strengths and weaknesses of our algorithms. As pointed out by Özen et al. [86], Markov decision pro-

cess based algorithms are slow in practice and cannot be used for big test instances. They develop a Markov decision process with a single state parameter, a single time step per period and one ordering decision per period. Using a binary search to find the optimal order, their algorithm already takes a few hours for a time horizon as small as $T = 18$ periods. For this reason, our simulations only consider a short time horizon $T \leq 15$, an aggregation of the time steps $R = 3$ and relatively small demand values. We set transportation delays as $L_{ord} = 1$ and $L_{del} = 2$, so that every period contains a time step before demand arrival and a time step with late demand. The asymmetry $L_{ord} = L_{del} - 1$ of the delays is a consequence of how we defined them. Indeed, suppose that a transport takes a little less that one time step. Empty containers depart at the beginning of a step and arrive before its end, while full packages depart during the step and arrive to the manufacturer during the next time step, hence after container ordering; finally we get $L_{ord} = 1$ but $L_{del} = 2$. We consider stationary costs, and write the vector of costs $C := [C_{man}, C_{sup}, C_{dis}, C_{setup}, C_{cont}]$. We model the demand variability as a binomial distribution $\mathcal{B}(n, p)$.

Table 7.2 shows the expected cost (rounded to an integer) and the running times (rounded to four digits) of the considered algorithms on very small instances. In this first simulation the maximum demand value $D_{\max}(T) := 3 \cdot T + 2$ increases linearly with T. As a consequence, even for $T = 5$ it takes more than 40 minutes to compute the optimal offline policy. We note that the online policy does not scale very well, due to the absence of a convexity structure. Up to now, we did not find any way to accelerate the computation without losing optimality. In this experiment, policies generated by Algorithm $Mdp.F$ are optimal or close to optimal. Moreover, our offline heuristic computes much better solutions than the simple algorithm 'NoDis'. However, the running time of our algorithm still increases much faster, and is not suited for instances where the demand can go above 50. We note that for $T = 3$, Algorithm $Mdp.F$ generates a suboptimal solution. Consequently, the best values u'_{k_1+1} and $k'_2(k_1)$ corresponding to the reference state $[0, 0, 0]$ are not equal to the optimal values $u^*_{k_1+1}$ and $k^*_2(k_1)$. There is a slight gain by using a quasi-offline policy instead of an offline policy. Moreover, the optimal and the heuristic quasi-offline policies are identical for the simulation data. We deduce that the reference state $[0, 0, 0]$ has little impact on the optimal value of the future placement. The alternative offline heuristic is as fast as the first offline heuristic.

Furthermore, the expected cost of the quasi-offline heuristic is not only the same as the optimal quasi-offline policy, but also the same as the quasi-online policy. Therefore, in our test instances the state of the system at purchasing time has little to no influence on the next purchasing time. In addition, both our quasi-online algorithm and quasi-offline heuristic should give a very tight bound on the expected cost of the optimal quasi-offline policy.

Table 7.2: Expected cost and running time of the algorithms for different time horizons. We use: $C := [2, 6, 50, 200, 50]$, $D(t, r) \to 3 \cdot t + \mathcal{B}(2, 1/2)$.

	$T = 3$	$T = 4$	$T = 5$	$T = 6$	$T = 7$	$T = 8$
$Mdp.N$ cost	1080	1946	2817	3832	5022	6335
$Mdp.N$ time	1s054	6s580	29s44	88s11	242s9	572s5
$Mdp.Qn$ cost	1080	1946	2817	3832	5022	6335
$Mdp.Qn$ time	0s515	2.699	10s06	26s60	66s94	148s0
$Opt.Qf$ cost	1080	1946	2817	3832	_	_
$Opt.Qf$ time	0s936	7s644	37s19	181s5	_	_
$Mdp.Qf$ cost	1080	1946	2817	3832	5022	6335
$Mdp.Qf$ time	0s312	1s279	3s931	8s159	17s28	32s71
$Opt.F$ cost	1080	1946	2816	(3832)	_	_
$Opt.F$ time	0s187	50s20	5907s	_	_	_
$Mdp.F$ cost	1080	1946	2816	3832	5035	6337
$Mdp.F$ time	0s047	0s187	0s484	0s967	2s043	3s604
$Mdp.F^{bis}$ time	0s047	0s156	0s406	0s967	1s934	3s588
$Ndis.F$ cost	2244	3300	4482	5880	7280	8822
$Ndis.F$ time	0s016	0s	0s	0s016	0s016	0s031

	$T = 9$	$T = 10$	$T = 15$
$Mdp.N$ cost	7679	9147	_
$Mdp.N$ time	1229s	2346s	_
$Mdp.Qn$ cost	7728	9147	_
$Mdp.Qn$ time	299s7s	563s7s	_
$Opt.Qf$ cost	(7728)	(9147)	_
$Opt.Qf$ time	_	_	_
$Mdp.Qf$ cost	7728	9147	17672
$Mdp.Qf$ time	56s86	93s57	618s4
$Opt.F$ cost	(7728)	(9147)	_
$Opt.F$ time	_	_	_
$Mdp.F$ cost	7728	9147	17672
$Mdp.F$ time	6s29	10s36	71s74
$Mdp.F^{bis}$ time	6s224	10s36	71s85
$Ndis.F$ cost	10366	12052	21352
$Ndis.F$ time	0s016	0s016	0s094

In the simulations, the offline, the quasi-offline and the quasi-online heuristic have very close value. This shows that the offline heuristic performs well for reasonable system cost. In such a case, the quasi-offline heuristic is not very interesting. However, Algorithm $Mdp.F$ may perform poorly when the optimal offline policy purchases new containers at nearly every period. Indeed, our reference state $[0, 0, 0]$ deciding on the best fleet size underestimates the required container fleet, because it neglects the value of X_t and Z_t at

Table 7.3: Expected cost of the different policies for different cost functions. We use $T = 4$, $D(t, r) \to 2 \cdot t + \mathcal{B}(2, 1/2)$.

	$Mdp.N$	$Opt.F$	$Mdp.F$	$Ndis.F$
Running time	1s763	95s01	0s094	0s
$C := [3, 6, 50, 100, 50]$	1816	1816	1816	2476
$C := [3, 6, 150, 10, 50]$	2046	2055	2136	2316
$C := [3, 6, 1500, 0, 50]$	3511	3547	3639	3636

	$Mdp.Qn$	$Opt.Qf$	$Mdp.Qf$
Running time	1s014	1s560	0s577
$C := [3, 6, 50, 100, 50]$	1816	1816	1816
$C := [3, 6, 150, 10, 50]$	2050	2051	2053
$C := [3, 6, 1500, 0, 50]$	3511	3511	3511

purchasing time. The approximation error increases when purchasing times get closer to each others. Table 7.3 presents a simulation with increasing disposable cost and decreasing setup cost, which leads to a worse and worse performance of Algorithm $Mdp.F$. In the last instance of this second simulation, with costs $C = [3, 6, 1500, 0, 50]$, the offline heuristic $Mdp.F$ gets high cost when purchasing at consecutive periods. In the simulation, the offline policy does not purchase any container at period 2, contrary to the other algorithms. We conclude that the reference state should be adapted to the system cost. On the opposite, the quasi-offline heuristic $Mdp.Qf$ always performs nearly as well as the optimal quasi-offline policy and the quasi-online algorithm.

We conclude from these simulations that our offline heuristics performs very well in general, but not in every scenario. When the purchasing times are close to each other, the offline policy does not compute the optimal purchasing size. Our quasi-offline heuristic seems to always perform close to optimal, and does not have the drawback of the offline heuristic. Moreover, the optimal quasi-online and quasi-offline policies have very similar expected costs.

7.8 Extensions and Outlook

In this chapter, we considered the stochastic version of the container management problem, where each demand is a random variable whose distribution is known at the beginning of the time horizon.
We model the stochastic problem as a Markov decision process, and consider four different strategies. The online strategy takes decisions as late as possible, so at period t we decide how many containers are purchased and how many are ordered during this period. The offline strategy assumes that the purchasing plan must be decided at the beginning of the time horizon,

whereas the order quantities are chosen dynamically. The quasi-offline strategy fixes the purchasing times at the beginning, but dynamically decides on the actual purchase quantities, along with the order size. The quasi-online strategies is similar to the online strategy, but fixes the next purchasing time in advance. The first three strategies are extensions of the basic policies proposed by Bookbinder and Tan [11], whereas the last one corresponds to the well-known Silver-Meal heuristic [106].

Instead of computing an optimal policy for each strategy, we approximate the solutions using a common dynamic programming framework. Each of these four algorithms runs in pseudo-polynomial time. The online algorithm is optimal but much slower than the others, due to not having a convexity property. The offline and quasi-offline algorithms are approximations, as they use a reference state to guess the best purchasing decision at every placement. Under this approximation, the cost functions are $L^\natural$-convex, which reduces the running time. The quasi-online policy is both optimal and has $L^\natural$-convex cost functions. This quasi-online algorithm justifies in a way the meaningfulness of the offline and quasi-offline heuristics, as all three algorithms use the same framework.

Simulations show that the heuristic algorithms perform well in general, but the offline algorithm has a rather poor quality in some extreme scenarios where our reference state is not adapted. The quasi-offline policy is less interesting in a container management problem than in a traditional inventory control problem. Despite this fact, our offline heuristic is very sensitive to the service level in an optimal solution, in contrast our the quasi-offline policy. Consequently, the quasi-offline solution measures to some extend the quality of the offline policy. Moreover, all algorithms are rather slow in practice, so further work is needed to get faster solutions. The simulations provided on Chapter 9 aggregate the demand to compute the policies.

Our Markov decision process model can be extended in several ways. Consider first a bi-criteria optimization problem, where the number of placement is a additional function to minimize. Since we are using a dynamic programming framework, we can adapt our algorithm to compute the best policies for every possible number of placements by adding a fourth parameter in the state relative to the number of placements. The total complexity increases by a factor of $O(T)$. Furthermore, we can extend our problem to variable lead times and variable number of time steps per period, as long as the container sent at a period arrive before the second next ordering time, i.e. as long as for every $t \in [0, T-2]$ and $r \in [0, R_t[$ we have $r + L_{del}(t, r) \leq R(t) + R(t+1)$. The model can also be extended to several suppliers, several manufacturers and longer lead times, but at the price of bigger decision spaces. Since the running time is already fairly slow for three state space, we cannot realistically compute any solution for higher state space without using some approximation method.

Chapter 8

Alternative Models

In this chapter, we have a look at other modelings and resolutions of $SCPP$. Firstly, we study the offline problem in two special cases. The first case considers a saturated system with more containers than needed. We then use the solution of this system to improve our choice of reference state. The second case assumes immediate lead time. Aferward, we discuss approximation algorithms which are efficient in classical inventory control and container repositioning problems. In particular, we consider approximate dynamic programming.

8.1 Saturated Offline Policy

In the deterministic model, we considered the special case where disposables are not allowed. In this section, we adapt this idea into the stochastic model and compute the best offline policy in this scenario.
Because the demand is stochastic, the demand may attain high values with very low probability. Nevertheless, disposables are very profitable for such unexpected demands. Thus, we should not forbid the use of disposables. In this section, we look for the best policy so that the order size is not restricted by the number of containers in the system instead. If this property holds, we say that the system is *saturated*. In a saturated system the order size β_t only depends on the supplier stock X_t. Indeed, the manufacturer stock Y_t is only considered in the system state as a limit of the order size. The outgoing fleet Z_t is added to the state parameters to remember the container fleet size. Consequently, the ordering policy can be computed independently from the container fleet size and hence from the purchasing plan. From the best ordering policy, we deduce the minimum number of containers which we must have in the system at every period to ensure that the system is saturated and that the ordering policy is feasible. The best purchasing plan respects these fleet size constraints at minimum cost.
In the following, we first compute the best ordering policy. Then we de-

duce the saturation levels and finally compute the best purchasing plan in a saturated environment.

8.1.1 Ordering Policy

In Chapter 7, we have restricted the maximum fleet size to value U_M:

$$U_M := (R + L_{ord} + L_{del}) \cdot D_{\max}$$

Therefore, the system is saturated if the container fleet size at period 0 is U_M.
We can compute the best ordering policy using a Markov decision process as for the online algorithm, but with a single parameter X_t per state, with a constant container fleet size U_M and without any purchasing decision.

Algorithm 23: Best Order Policy in a Saturated System

$\varphi^*_T(\boldsymbol{s}) := 0$ for all $\boldsymbol{s}$;
for *period* t *from* $T-1$ *to* 0 **do**
  for *step* r *from* $R-1$ *to* 1 **do**
    foreach *state* $\boldsymbol{s_{t,r}}$ **do**
      Compute $\varphi^*_{t,r}(\boldsymbol{s_{t,r}})$ using (7.12), (7.15) and (7.16);
  foreach *state* $\boldsymbol{s_t} = [x_t]$ **do**
    Compute the optimal order size $\beta^*_t(x_t)$
return policy β^*;

Since the order size is independent from both the outgoing fleet and the manufacturer stock, computing the ordering policy is equavalent to solving a standard stochastic inventory control problem with lost-sales and fractionnal lead times. The only difference to the model of Zipkin [141] is that we only order a quantity every R periods. The cost function are thus $L^\natural$-convex:

Lemma 8.1.1 *In a saturated environment, the functions $\psi_t(x_t)$ and $\varphi_t(x_t)$ are $L^\natural$-convex.*

Proof:
The proof is very similar to the one in Chapter 7 but simpler. We consider an alternative state space with two parameters corresponding to the supplier stock and inventory position. At time step $r = 0$, the decision variable is the inventory position after purchasing $x_t + \beta_t$ instead of $-\beta_t$. At time step $r = L_{ord}$, the inventory position becomes equal to the inventory level and the parameter relative to the inventory position will be equal to the inventory level until time $(t+1, 0)$. The cost function $\varphi_{t,r}$ is $L^\natural$-convex as it solves a minimization problem with decision $\delta_{t,r}$ and where the cost is the sum of expected values of $L^\natural$-convex functions.

□

Corollary 8.1.2 *In a saturated environment, the derivative of the optimal order size $\beta_t^*(x_t)$ is in $[-1, 0]$.*

Proposition 8.1.3 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. Algorithm 23 computes the best order policy in a saturated system in $O(T \cdot R^2 \cdot D_{\max}^2)$ time.*

Proof:

Algorithm 23 computes the best saturated policy because it computes the exact solution of the corresponding Markov decision process. By Lemma 7.3.1, the state space is in $O(R \cdot D_{\max})$, the decision space is in $O(R \cdot D_{\max})$, the demand space is in $O(D_{\max})$ and the time is $R \cdot T$. Since the cost functions are $L^\natural$-convex, it takes $O(\log[R \cdot D_{\max}] \cdot D_{\max})$ to compute the best order size and expected cost for state $[0, 0, 0]$ at period t. In addition, it takes $O(D_{\max})$ to compute the best order size and expected cost for state $[x, y, z]$ at period t given the best order size for either $[x-1, y, z]$, $[x, y-1, z]$ or $[x, y, z-1]$. We conclude that the time complexity of Algorithm 23 is $O(T \cdot R^2 \cdot D_{\max}^2)$.

□

8.1.2 Saturation Levels

For $t \in [0, T[$, we define the *(offline) saturation level* η_t as the minimum number of containers required in the system after purchasing at period t to ensure that the order size is not limited by the container fleet size for any demand scenario and policy before period t:

$$\forall t \in [0, T[,\ \eta_t := \max\left[x_t + z_t + \beta_t^*(x_t);\ \mathbb{P}\big(S_t = [x_t, U_M - x_t - z_t, z_t]\big) > 0\right] \tag{8.1}$$

Indeed, if the number of container after purchasing is at least η_t, then no matter where the containers are, we will have at least $\beta_t^*(x_t)$ containers in the manufacturer stock. Moreover, if we used $\eta_t - 1$ containers, there would be at least one state that can be attained with positive probability in which we cannot order $\beta_t^*(x_t)$ empty containers. The fsaturation level η_t is hence equal to the minimum fleet size.

However, we note that η_t is not necessarily increasing in t, because we have no constraint on the evolution of the demand distribution. We define the *cumulative saturation level* $\eta_{[0,t]}$ as the minimum fleet size required after purchasing at period t for the process to be saturated:

$$\eta_{[0,t]} := \max_{t \in [0,t]} \eta_t \tag{8.2}$$

If the container fleet size after purchasing at period t is lower than $\eta_{[0,t]}$, there is at least one previous period $t_o < t$ where the fleet size is lower than η_{t_o} and thus the system is not saturated. Reciprocally, if the container fleet

size after purchasing is at least $\eta_{[0,t]}$ at every period t, then the system is saturated. We deduce that the cumulative saturation levels are the fleet size constraints that the purchasing policy needs to respect.

To compute the saturation levels, we can use a forward search on the process using a state $\boldsymbol{s}_{t,r}$ with the two parameters $x_{t,r}$ and $z_{t,r}$. The parameter $z_{t,r}$ stores the number of outgoing containers if $r > R - L_{del}$ and the order size if $r < L_{ord}$. Therefore, a naive forward search computes the state probabilities in $O(T \cdot R^3 \cdot D^3_{\max})$ time.
Following (8.1) and (8.2), we compute the saturation levels from the state probabilities in $O(T \cdot R^2 \cdot D^2_{\max})$ time and deduce the cumulative saturation levels in $O(T)$ time.

Proposition 8.1.4 *Using a forward dynamic program, we can compute the cumulative saturation levels for any ordering policy in $O(T \cdot R^3 \cdot D^3_{\max})$ time.*

In fact, we can compute them faster by making use of Corollary 7.4.8. When we do not consider setup cost, the optimal order size decreases by either zero or one when either x_t, $-y_t$ or $-z_t$ increases by one. This property holds for both saturated and non-saturated systems.

Lemma 8.1.5 *Let $x^*_{t-1,R-L_{del}+1}$ be the maximum supplier sotck at time $(t-1, R-L_{del}+1)$ given the optimal saturated order policy. Then η_t equals the required fleet size at period t when starting from supplier stock $x^*_{t-1,R-L_{del}+1}$ and having a maximal late demand at time $(t-1, r)$, $r \in]R - L_{del}, R[$.*

Proof:

By definition, η_t is the maximum value of $x_t + z_t + \beta^*_t(x_t)$. Consider the supplier stock $X_{t-1,R-L_{del}+1}$ before the first late demand. Then every container leaving the supplier stock until period t will be outgoing at ordering time, so we have:

$$X_{t-1,R-L_{del}+1} = X_t + Z_t \tag{8.3}$$

Consider a supplier stock $x_{t-1,R-L_{del}+1}$ at time $(t-1, R-L_{del}+1)$ and a realization of the total late demand at period $t-1$. They induce the parameter values $X_t = x_t$ and $Z_t = z_t$. We can control the value of z_t with the total late demand. If z_t increases by one, then x_t decreases by one and hence β^*_t increases by zero or one. Therefore, the required fleet increases with z_t. We deduce that for a fixed supplier stock $x_{t-1,R-L_{del}+1}$ at time $(t-1, R-L_{del}+1)$, we obtain the maximum required fleet when every late demand is maximal.
Morevoer, when we increase $x_{t-1,R-L_{del}+1}$, either x_t or z_t increases by one and thus β^*_t decreases by zero or one. Consequently, the required fleet size is non-decreasing in $X_{t-1,R-L_{del}+1}$ and the required fleet η_t is attained when starting from stock $x^*_{t-1,R-L_{del}+1}$.

□

From the maximum supplier stock at time $(t, R-L_{del}+1)$, we hence compute η_t in $O(R \cdot D_{\max})$ time, assuming that we can access the maximum demand at each time step in constant time.

Lemma 8.1.6 *Let $D^{min}_{t,0:L_{ord}-1}$ be the minimum total demand between the ordering time $(t,0)$ and the order arrival time (t, L_{ord}). We can choose β^*_t so that:*

$$\forall x_t \leq D^{min}_{t,0:L_{ord}-1} : \ \beta^*_t(x_t) = \beta^*_t(0) \tag{8.4}$$

Proof:

If the supplier stock x_t at ordering time is lower than the minimum demand during the ordering delay, then the supplier stock at order arrival is equal to the order size $\beta^*_t(x_t)$. Therefore, the expected cost function after order arrival is the same for every x_t $in [0, D^{min}_{t,0:L_{ord}-1}]$. We can thus choose the same order size minimizing this function.

□

Proposition 8.1.7 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. We can compute the cumulative saturation levels for any ordering policy in $O(T \cdot R \cdot D_{\max})$ time.*

Proof:

We have seen in the previous chapter that without any setup cost the derivative of the order size function $\beta^*_t(x_t)$ with respect to x_t only has values in $[-1, 0]$.

□

In the special case where the support of each demand is an interval, we can reduce the computation time to calculate the saturation levels. We first compute the possible values of the supplier stock $X_{t,R-L_{del}+1}$, in $O(T \cdot R^2 \cdot D^2_{\max})$. If we can get the minimum and maximum demands in constant time, then we compute in $O(T \cdot R)$ time the minimum and the maximum total late demand at every period. Otherwise, it takes $O(T \cdot R^2 \cdot D_{\max})$ time instead. Then, for each of the $O(R \cdot D_{\max})$ possible supplier stocks and $O(R \cdot D_{\max})$ possible total late demands, we compute the number of containers required at the next ordering $t+1$.

Proposition 8.1.8 *If the support of every late demand is an interval, we can compute the cumulative saturation levels for any ordering policy in $O(T \cdot R^2 \cdot D^2_{\max})$ time.*

8.1.3 Optimal Purchasing

We now compute the best purchasing policy making the system saturated, i.e. ensuring that the container fleet size after purchasing at period t is at least $\eta_{[0,t]}$.

Proposition 8.1.9 *Given the cumulative saturation levels, the problem of computing an optimal offline purchasing policy can be reduced to a dynamic lot-sizing problem over T periods with, at period $t \in [0, T[$:*

- *demand $D_{ww}(t) := \eta_{[0,t]} - \eta_{[0,t-1]}$ if $t > 0$, $D_{ww}(0) = \eta_0$.*
- *Purchasing setup cost $C_{setup}(t)$ and unit cost $C_{cont}(t)$.*
- *holding cost $C_{idle}(t)$*

Proof:

After purchasing at period t, we need $\eta_{[0,t]}$ containers in the system, and any additional container will necessarily incur the idleness cost $C_{idle}(t)$. Therefore, we want to purchase containers so that the container need increases by $D_{ww}(t)$ at period t, and we penalize with holding costs $C_{idle}(t)$ the containers purchased too early. This corresponds to the lot-sizing problem from this proposition.

□

Corollary 8.1.10 *Given the cumulative saturation levels, an optimal offline saturated purchasing policy can be computed in $O(R \cdot T)$ time.*

Proof:

The lot-sizing problem is solved by algorithms from the literature [27, 127, 1] in $O(T)$ time, and we need $O(T \cdot R)$ time to reformulate the container purchasing problem as a dynamic lot-sizing problem. Thus, it takes $O(R \cdot T)$ time to compute an optimal offline saturated policy given the cumulative saturation levels $\eta_{[0,t]}$.

□

Theorem 8.1.11 *Suppose that Hypotheses 2.1 [Delay] and 2.2 [Cost] hold. An optimal offline saturated policy can be computed in $O(T \cdot R^3 \cdot D_{\max}^3)$ time.*

8.1.4 New Reference State

In the offline heuristic, at every placement we used a reference state $[0, 0, 0]$ to estimate the best next placement and the best purchase size. However, this reference state is not adapted to test instances where placements are close to each other and the disposables are expensive. In this case, we need a reference state with a high supplier stock x_t and a high outgoing fleet z_t. As an alternative reference state, we consider the state $[x_t^{sat}, 0, z_t^{sat}]$ maximizing the saturation level at placement t. The best saturated policy is much faster than the offline policy, so computing the best saturated offline policy before using our offline heuristic will not noticeably increase the running time. We denote by $Mdp.F^{ref}$ the offline algorithm using this new reference state.

8.1.5 Simulations

We end this section with two experiments on the offline algorithms. Firstly, we compare the saturated policy to Algorithm $Mdp.F$ and the best policy without disposables. Secondly, we compare two variants $Mdp.F$ and $Mdp.F^{ref}$ of our offline heuristic using the initial reference state $[0, 0, 0]$ and the new one $[x_t^{sat}, 0, z_t^{sat}]$. Table 8.1 and 8.2 show the running times and expected costs of the algorithms using the the same simulation instances as in Chapter 7.

Table 8.1: Comparison of the expected cost and running time of the saturated policy and the updated offline heuristic $Mdp.F^{ref}$ to the algorithms from Chapter 7. We use: $C := [2, 6, 50, 200, 50]$, $D(t, r) \to 3 \cdot t + \mathcal{B}(2, 1/2)$.

	$T = 3$	$T = 4$	$T = 5$	$T = 6$	$T = 7$	$T = 8$
$Mdp.F$ time	0s047	0s187	0s484	0s967	2s043	3s604
$Mdp.F$ cost	1080	1946	2816	3832	5035	6337
$Mdp.F^{ref}$ cost	1080	1946	2816	3832	5035	6337
$Sat.F$ cost	2052	3062	4127	5478	6831	8325
$Sat.F$ time	0s005	0.007s	0.018s	0s029	0s044	0s057
$Ndis.F$ time	0s016	0.007s	0s	0s016	0s016	0s031
$Ndis.F$ cost	2244	3300	4482	5880	7280	8822

	$T = 9$	$T = 10$	$T = 15$
$Mdp.F$ time	9s553	16s00	114s7
$Mdp.F$ cost	7728	9147	17664
$Mdp.F^{ref}$ cost	7728	9147	17664
$Sat.F$ time	0s067	0s111	0s378
$Sat.F$ cost	9822	11461	20525
$Ndis.F$ time	0s077	0s016	0s094
$Ndis.F$ cost	10366	12052	21352

In the simulation presented in Table 8.1, the offline heuristic has the same expected cost for both choices of reference state. The two variants have the same purchasing plan. Thus, in most test instances, we can expect the offline heuristic with the new reference state to perform as well as using the initial reference state $[0, 0, 0]$.
In Table 8.2, there is a significant improvement of the expected cost when using the new reference state. For the second cost structure, the heuristic using the new reference state is nearly optimal. However, the variant does not entirely remove the problem as the difference of cost to the optimal solution only decreases by 33% for the third cost structure.
Furthermore, the best saturated policy performs much worse than the offline heuristics. Its performance is actually close to the best policy forbidding disposables. This results confirms the supposition we made in the previous chapter. Namely, the variance of the considerd demand distributions is small

Table 8.2: Comparison of the expected cost of the saturated policy and the updated offline heuristic $Mdp.F^{ref}$ to the algorithms from Chapter 7, when the initial offline heuristic $Mdp.F$ performs worse than usual. We use $T = 4$, $D(t, r) \to 2 \cdot t + \mathcal{B}(2, 1/2)$.

	$Opt.F$	$Mdp.F^{ref}$	$Mdp.F$	$Sat.F$	$Ndis.F$
Running time	95s01	0s094	0s094	0s005	0s
$C := [3, 6, 50, 100, 50]$	1816	1816	1816	2326	2476
$C := [3, 6, 150, 10, 50]$	2055	2058	2136	2190	2316
$C := [3, 6, 1500, 0, 50]$	3547	3609	3639	3611	3636

enough so that the best policy without disposables is close to the best policy we can attain if we always want to keep the same service level. Our conlusion is hence that we can significantly reduce the expected cost of the process if we use more disposables shortly before a placement. By doing so we reduce the manufacturer holding cost between two consecutive placements.

8.2 Offline Solution under Zero Delay

In this section we consider the simple case without lead time, which we have done in Chapter 5 but for deterministic demand:

Hypothesis 5.1 [NoDelay]: *The ordering and delivery delays are immediate, i.e.* $L_{ord} = 0$ *and* $L_{del} = 1$.

The consequence of immediate delivery is that there are never outgoing containers at the beginning of any time step. Therefore:

$$\forall t, \forall r: \ Z_{t,r} := 0$$

We denote the states at ordering as:

$$S_t := [X_t, U_t],$$

where the parameter $U_t = X_t + Y_t$ replaces Y_t. The time complexity of every algorithm from Chapter 7 is decreased by $O(R \cdot T)$. Moreover, systems with immediate ordering provide a good control on the supplier stock. We denote by ω_t the supplier stock after ordering.
In the following, we propose a faster offline algorithm under the hypothesis that the distribution demand is *stochastically non-decreasing*:

Hypothesis 8.1 ***[Demand-Sto]*** *The demand is stochastically non-decreasing in* t*:*

$$\forall t \in [1, T[, \ \forall d \in \mathbb{R}^+: \ \mathbb{P}(D_{t,r} > d) \geq \mathbb{P}(D_{t-1,r} > d) \tag{8.5}$$

We call *decision cost* at period t the total cost incurred from:

1. the supplier holding cost during period t.
2. the disposable cost during period t.

We call $\psi_t^{\omega}(w_t, u_t)$ the decision cost at period t when the post-purchasing supplier stock is ω_t, and $\varphi_t^{\omega}(\omega_t, u_t)$ the minimum total decision cost from period t to EoH when ordering the optimal quantities afterward. We also define, when starting with state $s_t = [x_t, u_t]$ at period t:

$$\varphi_t(x_t, y_t) := \min_{\beta \leq y_t} \Big[\varphi_t(x_t + \beta, u_t)\Big] \tag{8.6}$$

Lemma 8.2.1 *Under Hypotheses 2.2 [Cost] and 5.1 [NoDelay], the cost functions $\psi_t(\omega, u)$ and $\varphi_t(\omega, u)$ are $L^\natural$-convex for a fixed offline purchasing plan.*

Proof:

Similar to the $L^\natural$-convexity under zero setup cost from Chapter 7.

□

Corollary 8.2.2 *Under Hypotheses 2.2 [Cost] and 5.1 [NoDelay], the cost function $\varphi_t(\omega, u)$ is non-decreasing in $\omega \leq u$ for a fixed u.*

Proof:

When the starting supplier stock is $x_t = 0$, we can choose any post-ordering supplier stock without cost. Therefore, $x_t = 0$ is a minimum of the $L^\natural$-convex functions. By convexity, the cost function φ_t is then non-decreasing starting from $x_t = 0$ for all $t \in [0, T[$.

□

Given the fleet size u, we denote by $\omega_t^*(u)$ the smallest optimal supplier stock after order arrival in this saturated environment:

$$\omega_t^*(u) := arg\min_x \Big[\varphi_t(x, u)\Big] \tag{8.7}$$

Moreover, consider ω_t^{ψ} minimizing the single period decision costs:

$$\omega_t^{\psi}(u) := arg\min_x \Big[\psi_t(x, u)\Big] \tag{8.8}$$

Lemma 8.2.3 *If Hypotheses 2.2 [Cost] and 5.1 [NoDelay] hold, then we have:*

$$\forall t \in [0, T[:\ \forall u:\ \omega_t^*(u) \leq \omega_t^{\psi}(u) \tag{8.9}$$

Furthermore, if $x_t > \omega_t^*(u)$, then the optimal order size is zero.

Proof:

Suppose that at period t we have stock x_t and order β_t so that:

$$x_t + \beta_t > \omega_t^{\psi}(u)$$

The cost at period t is then not lower than $\psi_t^{\omega}(\omega)$. In addition, the supplier stock at period $t+1$ will be greater than the post-ordering supplier stock ω. Therefore, by Corollary 8.2.2, the cost starting from period $t+1$ is greater than if we only ordered up to $\omega_t^{\psi}(u)$. We conclude that if $x_t \leq \omega_t^*$, we should order up to $\omega_t^{\psi}(u)$. By Corollary 8.2.2, the cost starting from period $t+1$ is also not lower than if we ordered up to $\omega_t^{\psi}(u)$. We conclude that we should raise the inventory position at period t above $\omega_t^{\psi}(u)$

□

Lemma 8.2.3 states that the best order size should not make the inventory position exceed the myopic order-up-to quantity $\omega_t^{\psi}(u)$. Under a demand which is not stochastically non-decreasing, an optimal policy raises the inventory level below $\omega_t^{\psi}(u)$. For instance, if the demand at the coming period $t+1$ is expected to be very low, we should order less at period t because the leftover containers at the end of the period will likely stay in the supplier stock up to the end of period $t+1$. This cannot occur under stochastically non-decreasing demand:

Proposition 8.2.4 *Under Hypotheses 2.2 [Cost], 5.1 [NoDelay], and 8.1 [Demand-Sto], we have:*

$$\forall t \in [1, T[, \forall u: \ \omega_{t-1}^{\psi}(u) \leq \omega_t^{\psi}(u) \tag{8.10}$$

Proof:

Suppose that the fleet size u_t at period t is fixed. By definition, $\omega_t^{\psi}(u_t)$ minimizes:

$$\psi_t(\omega, u_t) := \psi_t^{dis}(\omega, u_t) + \psi_t^{sup}(\omega, u_t)$$

where:

$$\psi_t^{dis}(\omega, u_t) := \mathbb{E}_{\{D_{t,r}\}}\Big[\sum_{r=0}^{R-1}\big(C_{dis}(t,r) - C_{man}(t,r)\big)\cdot\max\big(\sum_{r'=0}^{r} D_{t,r'} - \omega\big)^+\Big]$$

$$\psi_t^{sup}(\omega, u_t) := \mathbb{E}_{\{D_{t,r}\}}\Big[\sum_{r=0}^{R-1} C_{sup}(t,r)\cdot\max\big(\omega - \sum_{r'=0}^{r} D_{t,r'}\big)^+\Big]$$

The derivative is zero for $\omega = \omega_t^{\psi}$.

$$\frac{\partial}{\partial\omega}\big\{\psi_t^{dis}(\omega, u_t)\big\}$$

$$= \mathbb{E}_{\{D_{t,r}\}}\Big[\sum_{r=0}^{R-1}\big(C_{dis}(t,r) - C_{man}(t,r)\big)\cdot\frac{\partial}{\partial\omega}\big\{\sum_{r'=0}^{r} d_{t,r'} - \omega\big\}^{+}\Big]$$
$$= -\sum_{r=0}^{R-1}\big(C_{dis}(t,r) - C_{man}(t,r)\big)\cdot\mathbb{P}\big(\sum_{r'=0}^{r} d_{t,r'} > \omega\big)$$

Likewise:

$$\frac{\partial}{\partial\omega}\big\{\psi_t^{sup}(\omega,u_t)\big\} = +\sum_{r=0}^{R-1} C_{sup}(t,r)\cdot\mathbb{P}\big(\sum_{r'=0}^{r} d_{t,r'} \leq \omega\big)$$

Thus:

$$\sum_{r=0}^{R-1}\mathbb{P}\big(\sum_{r'=0}^{r} d_{t,r'} \leq \omega_t^{\psi}(u_t)\big)\cdot\big(C_{sup}(t,r) + C_{dis}(t,r) - C_{man}(t,r)\big)$$
$$= \sum_{r=0}^{R-1}\big(C_{dis}(t,r) - C_{man}(t,r)\big)$$

Let $\mathbb{F}_{0:r}$ be the cumulative distribution function of $\sum_{r'=0}^{r} D_{t,r}$. We define:

$$C_{t,r}^{\omega} := C_{sup}(t,r) + C_{dis}(t,r) - C_{man}(t,r),$$
$$C^{\omega} := \sum_{r=0}^{R-1} C_{t,r}^{\omega}$$
$$\text{and } C_{qtl}^{\omega} := \frac{\sum_{r=0}^{R-1} C_{dis}(t,r) - C_{man}(t,r)}{C^{\omega}} \in [0,1]$$

Moreover, we define the function $\mathcal{F}^{\omega}$ so that we have for each x that:

$$\mathcal{F}^{\omega}(x) := \sum_{r=0}^{R}\frac{C_{t,r}^{\omega}}{C^{\omega}}\cdot\mathbb{F}_{0:r}(x)$$

As a weighted sum of cumulative distribution functions $\mathcal{F}^{\omega}$ itself is a cumulative distribution function. The optimal post-ordering stock ω_t^{ψ} is thus the C_{qtl}^{ω} quantile of this distribution.

$$\omega_t^{\psi}(u_t) = \min\Big\{\Big[\sum_{r=0}^{R}\frac{C_{t,r}^{\omega}}{C^{\omega}}\cdot\mathbb{F}_{0:r}\Big]^{-1}\big(C_{qtl}^{\omega}\big),\ \ u_t\Big\} \tag{8.11}$$

By hypothesis, the demand is stochastically non-decreasing, consequently $\mathbb{F}_{0:r}$ is increasing in t for each $r \in [0,R[$. Since $\mathcal{F}^{\omega}$ is an increasing function of the cumulative distribution functions $\mathbb{F}_{0:r}$ and due to u_t being non-decreasing, it follows that $\omega_t^{\psi}(u_t)$ is non-decreasing in t.

□

Theorem 8.2.5 *Supper that Hypotheses 2.2 [Cost], 5.1 [NoDelay], and 8.1 [Demand-Sto] hold. Then* $\omega_t^*(u_t) = \omega_t^{\psi}(u_t)$ *minimizes* $\psi_t(\cdot, u_t)$.

Proof:

Firstly, the optimal stock $\omega_{T-1}^*(u_{T-1})$ after ordering minimizes

$$\psi_{T-1}^{\omega}(\cdot, u_{T-1}) = \varphi_{T-1}(\cdot, u_{T-1}).$$

Consider $t \in [0, T-2]$ and suppose that $\omega_{t+1}^*(u_{t+1})$ minimizes $\psi_{t+1}(\cdot u_{t+1})$. By Proposition 8.2.4, the stock ω_t^{ψ} minimizing ψ_t is so that:

$$\omega_t^*(u_t) \leq \omega_t^{\psi}(u_t) \leq \omega_{t+1}^{\psi}(u_{t+1}) = \omega_{t+1}^{\varphi}(u_{t+1})$$

Therefore, we have $\omega_t^*(u_t) \leq \omega_{t+1}^*(u_{t+1})$. Consequently, there is no advantage anymore in having less than $\omega_t^{\psi}(u_t)$ containers in the supplier stock after ordering. In fact, this would increase the decision cost at period t without decreasing the cost after period t, as the leftover stock at period $t+1$ will not be greater than the desired quantity $\omega_{t+1}^*(u_{t+1})$. We deduce:

$$\omega_t^*(u_t) = \omega_t^{\psi}(u_t)$$

The proposition follows by recurrence on t.

□

By Theorem 8.2.5, we can compute each cost function $\psi(\omega_t, u_t)$ independently from each other. We deduce the optimal purchasing policy from these values. We define the locally optimal policy cost $\psi[k_1, k_2, u]$ between consecutive placements k_1 and $k_2 - 1$ given that the container fleet size after purchasing at period k_1 is u. The value $\psi[k_1, k_2, u]$ is equivalent to a value " $H(k_1, k_2)$ "in the W-W algorithm. We define $u^*(k_1, k_2)$ so that:

$$H^*(k_1, k_2, u^*(k_1, k_2)) = \min_u \left[H^*(k_1, k_2, u)\right] \tag{8.12}$$

Algorithm 24: Offline Algorithm under Zero Delay

```
for period t : 0 → T − 1 do
    foreach Fleet size u_t do
        Compute ω*_t(u_t);
        Compute ψ_t(ω*_t(u_t), u_t);
for period k_2 : T → 1 do
    for period k_1 : 0 → k_2 − 1 do
        foreach fleet size u do
            Deduce from the ψ_t(ω*_t(u), u) the cost ψ[k_1, k_2, u];
        Deduce u*(k_1, k_2);
    Deduce φ*_{k_2};
return φ*_0;
```

Theorem 8.2.6 *Suppose that Hypotheses 2.2 [Cost], 5.1 [NoDelay] and 8.1 [Demand-Sto] hold. Then Algorithm 24 computes an optimal offline policy in* $O(T^2 \cdot R \cdot D_{\max} + T \cdot R^2 \cdot D_{\max}^2)$ *time.*

Proof:

Using backward programming, computing the functions ψ_t takes $O(T \cdot R^2 \cdot D_{\max}^2)$ time, because the $R-1$ iterations for $r > 0$ can be summarized to a single parameter state $S_{t,r} = [X_{t,r}]$, and the last iteration $r = 1$ can be summarized with state $S_t = [u_t]$ by Theorem 8.2.5. The $\psi[\cdot,\cdot,\cdot]$ functions can be computed in $O(T^2 \cdot R \cdot D_{\max})$ time. Indeed, for each k_2 and every u, we can compute every $\psi[k_1, k_2, u]$ together in $O(T)$ time, by computing directly:

$$\psi[k_1, k_2 + 1, u] = \psi[k_1, k_2, u] + \phi_{k_2+1}(\omega_t^*(u), u)$$

□

Remark 8.2.7 *Under zero delay and for stochastically non-decreasing demands, the optimal offline policies are also optimal as a quasi-offline strategy. In fact, by Theorem 8.2.6 and Proposition 8.2.4, we can order the containers so that every container ordered at period* t *are also ordered at period* $t+1$*. Since there is no ordering cost, the cost up to EoH when starting at placement* k *and state* $[x_k, z_k]$ *is equal to the cost at state* $[x'_k, z'_k]$*, where:*

$$x'_k = \min\{\omega_t^*,\ x_k + z_k\}$$
$$z'_k = x_k + z_k - x'_k$$

Therefore, the cost and the purchasing decision at placement k *do not depend on the state at period* k*.*

8.3 Other Resolution Approaches

In this section, we present and comment on a literature overview of other ways of solving the stochastic container purchasing problem. First and foremost, we review the framework of *approximate dynamic programming* (ADP), which we believe is very efficient to compute good policies and can be used jointly with our resolution approach from Chapter 7. Then, we very briefly present dual-balancing policies and discuss meta-heuristics.

8.3.1 Approximate Dynamic Programming

Introduction

Whenever we use a Markov decision process to solve a problem, we are bound to face the *curse of dimensionality*: the running time explodes whenever the state space and the decision space are not very limited.

A popular way to decrease the running time while keeping a good solution quality is to approximate the cost functions φ_t using approximate dynamic programming. Powell [90] dedicates a book to ADP, which covers the concepts, the mathematical foundations as well as detailed applications in logistics. The author also provides a succinct introduction to ADP in [89]. As described by Meisel [75], the two main alternatives to approximate the functions φ_t are *state space aggregation*, decreasing the state space, and *predictive modeling*, simplifying them to a parametrized function. Meisel [75] proposes to use business intelligence to accelerate an approximate dynamic program.

State Space Aggregation

State space aggregation consists in computing the cost value and decision only for a few states which we call *main states*. We call the other states *aggregated states*. We implicitly approximate the decision and the cost value at the aggregated states using the decision and cost value of the main states in some neighborhood. The simplest aggregation method uniformly aggregates the state space. It considers an aggregation factor N_i for each state state parameter i. Every state whose value of parameter i is a multiple of N_i for every i is chosen as a main state. In addition to aggregation of the state space, we can also aggregate the demand and the decision space. We use nonetheless the term of "state"space aggregation because it is the most commonly used in the literature.
Halman et al. [40, 39] use a smart form of state space aggregation to compute their fully polynomial time approximation scheme. They exploit the convexity of their problem to show that the implicitly computed values are close enough to their actual values, so that the final solution has a relative performance guarantee. Likewise, Chen et al. [14] use a coarse cube structure to show that their pseudo-polynomial time approximation scheme has an absolute performance guarantee. Arts et al. [5] make another form of aggregation. They consider an inventory control model where the lead time takes several periods, hence the state contains a list of L outgoing orders. They aggregate the outgoing orders into a single sum and prove that the next order arrival can be approximated well enough to generate an asymptotically optimal solution for high lost-sales cost.
There are many ways to aggregate our container management process. On the one hand, we can aggregate the time steps to only have three time steps per period, corresponding to preliminary, early and late time steps. One the other hand, we can aggregate the state space, the decision space, and the demand space, for example by adapting the approximation scheme from Chen et al. [14].
For each type of aggregation, we need to create a formula to approximate the cost value and decision of the aggregated states.

Predictive Modeling

In predictive modeling we approximate either the cost value of being in a state or the corresponding decision to take.
The method of *value approximation* is the most widely used method. It consists in approximating the cost functions φ_t. There are many ways of approximating the functions φ_t. A common method uses *Monte Carlo simulations*. This is a simple framework where we iteratively generate *Monte Carlo samples* which are random scenarios of demand realization starting form an initial state. Each sample is used to update the quality of the solution, namely the cost of being in each state. Monte Carlo simulations enable to update a reduced number of states for each sample. There are several methods to update the cost functions like using Q-learning or the temporal difference learning $TD(\lambda)$. This approach has been proven to converge to the optimal solution. As an alternative approach, we approximate each function φ_t using a parametric function. In this case, we only need to compute the value at a few points to estimate the parameters and hence get an approximation of the cost of being in every state. This approach significantly reduces the running time but requires more knowledge on the cost structure to find a good parametrized function to optimize.
The method of *policy approximation* is used if we are able to approximate the value of being in the future states and represent the policy using a parametric function. We then compute the optimal ordering decision for several states to estimate the best value for the parameters. In our framework, we need to compute the value of being in each state in order for the dynamic program to make the purchasing decisions.
Besides, a myopic ordering policy would optimize the cost over a short time horizon. In this case, we recommend at least two periods of time horizon to take into account the limitation of the next order size due to the outgoing fleet. We expect an ordering policy optimizing the within one single period of time to perform poorly.

Applications of ADP

Approximate dynamic programming has been successfully applied to several logistic problems, such as in Powell and Topalogu [91], Lam et al. [66], Powell [90] and Meisel [75]. The researchers in this field insist on the fact that in most cases, approximate dynamic programming and especially predictive modeling are meaningless without a good understanding of the problem and the solution properties. Indeed, a poorly chosen parametrized function may lead to a very poor performance.
Therefore, some properties such as an asymptotic behavior, convexity of the costs and an experimental analysis of the optimal solution are a good way to start a study before deciding on the ADP.

For instance, Huh et al. [50] prove that the best base-stock policy is asymptotically optimal in a inventory control system with lost-sales when the shortage cost gets large. Therefore, we can use a base-stock policy to approximate the optimal order size. Arts et al. [5] consider the same problem when the lead time L_{ord} is not negligible. Consequently, the state consists in an inventory level and $L_{ord}-1$ outgoing order sizes. They approximate this state with only two parameters, namely the inventory level and the sum of the outgoing orders. They approximate the number of items arriving at time t from the distribution of the demand at time $t - L_{ord} - 1$ and the sum of outgoing orders. They prove that when the shortage cost gets large, the best policy relative to this approximate state representation is asymptotically optimal. An ADP using this space aggregation has $L_{ord} - 2$ fewer parameters, with is a significant improvement.
For the same problem but for infinite time horizon, Xin and Goldberg [133] and Goldberg [32] show that a constant order policy is asymptotically optimal when the lead time gets large. Consequently, when the lead times are long, we have an easy way to approximate the optimal solution.

Possible Applications in $SCPP$

Firstly, we can use an aggregation, in particular when the demand realizations are big. In Chapter 9, the containers are aggregated in pallets of 15.
Secondly, we may use a parametric approximation of the optimal order size. In the algorithms from Chapter 7, the cost functions are $L^\natural$-convex, so by Corollary 7.4.8, the gradient of the cost functions is between -1 and $+1$. This behavior is similar to the traditional inventory control problems with lost-sales. Consequently, we expect the optimal ordering policy in our container management problem to be close to a restricted base-stock policy.
An interesting approach for future research consists in approximating at each period the order size $\beta_t(x_t, y_t, z_t)$ with:

$$\beta_t(x_t, y_t, z_t) \approx \min\{S_t - x_t, y_t^\alpha, R_t^1(x_t), R_t^2(u_t^\alpha)\}, \tag{8.13}$$

where S_t is the base-stock level, y_t^α the manufacturer inventory level after purchasing, $R_t^1(x_t)$ an upper-bound of the order size in a saturated environment, and $R_t^2(u_t^\alpha)$ an upper-bound of the order size given the post-purchasing container fleet size u_t^α. The parameter $R_t^1(x_t)$ hence corresponds to the second parameter R in a restricted base-stock policy. Meanwhile, the parameter $R_t^2(u_t^\alpha)$ represents a new constant specific to container management problems with a limited fleet size, and should avoid ordering too many containers at a period and as a result not having enough containers for the next period. Similarly to the vector base-stock policy from Zipkin [142], we can have S_t, R_t^1 and R_t^2 depend on a same parameter θ_t approximating a service level.

A third idea would be to approximate the cost functions φ_t with a second degree polynomial and to estimate its three parameters with a approximation of the expected cost at some states.

8.3.2 Dual-Balancing Policies

Dual-balancing policies are a simple and efficient way of computing near-optimal online policies when the system is composed of concurring cost functions.

Even though they were already used before, dual-balancing policies were introduced in the inventory control literature and first analyzed by Levi et al. [70]. They consider a periodic-review stochastic inventory control problem with positive lead times, holding cost and backlogging cost. They define the *marginal cost* of a decision as the expected cost generated by this decision. The marginal cost regroups the shortage cost from the order arrival to the next order arrival and the holding cost of the ordered items up to the end of the time horizon, assuming that the items are sold in the same sequence as they are ordered. The authors prove that such a policy has a performance guarantee of 2, i.e. the cost of this policy is at most twice the optimal expected cost. They extend their study to include a setup cost and show that the resulting policy has a performance guarantee of 3.

Hurley et al. [51] extend these policies to include lower and upper bounds on the order sizes. They also conduct an extensive computational study of these policies under a *Martingale Model of Forecast Evolution (MMFE)*. The MMFE is a popular framework to model the evolution of forecasts, and has been introduced by Heath and Jackson [45]. Levi et al. [71] extend the study without setup cost to capacitated systems facing non-stationary and correlated demands. The capacities define a maximum order size at each period. The authors generalize the dual-balancing policy by adding to the martingale cost a forced backlogging cost representing additional items one could have ordered and will not be able to order in the future due to the order capacity. Levi et al. [69] extend these study to lost-sales models, and prove that the inventory control problem without setup cost has a performance guarantee of 2. Shi et al [105] propose a 4-approximation generalizing the inventory models with backlogged demand with setup cost, correlated demand, capacitated ordering, and batch ordering[1]. Shi [104] presents an overview of these approximation algorithms. We note that only Levi et al. [69] study a system with lost sales.

We find the idea of a dual balancing policy very interesting, and it is an open question whether there is a dual-balancing policy for the online and quasi-online strategies with a performance guarantee.

[1] i.e. where the ordering is a multiple of a constant integer.

8.3.3 Meta-Heuristics

There are many meta-heuristics in the literature such as the genetic algorithms, the simulated annealing algorithms and the tabu search algorithms to cite a few. These local searches successively compute policies and update the best computed policy form each policy cost.
However, using our Markov decision process it already takes $O(T \cdot D_{\max}^4 \cdot R^4)$ time to compute the cost of an optimal policy. Indeed, we iterate over $O(T \cdot R)$ time steps, with $O(D_{\max}^3 \cdot R^3)$ state and $O(D_{\max})$ possible realizations of the demand per time step. Consequently, meta-heuristics may not be significantly faster than algorithms based on Markov decision processes.
Furthermore, classical approximation methods to evaluate the policy cost such as state and demand aggregation or Monte-Carlo simulations can also be applied to the algorithms covered in the previous chapter. Therefore, we feel that most meta-heuristics are not a meaningful alternative to the algorithms presented in Chapter 7.

Part III

Application and Conclusion

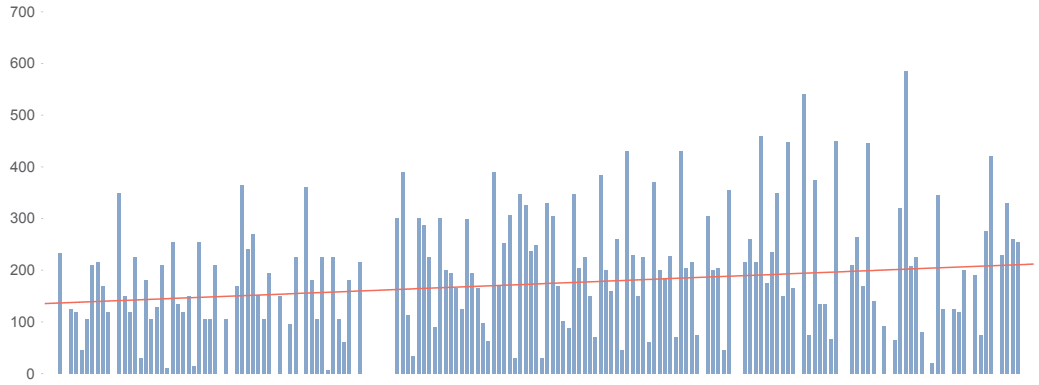
Chapter 9

A Real-Life Application

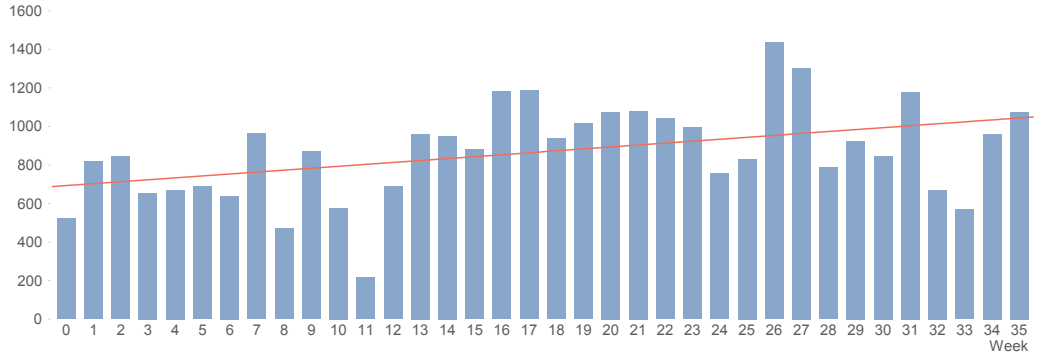
In this chapter, we apply the offline algorithms of this thesis to the closed-loop supply chain which we described in Chapter 1. In Section 9.1, we estimate every demand distribution from the project data with a regression model. Section 9.2 presents our simulation environment. Finally, we present and discuss the experiments results in Section 9.3.

9.1 Problem Data

The items are transported in KTL containers, and the containers are packed in pallets of fifteen. We consider a time horizon of $T = 36$ weeks. One time step represents a working day and one period is a week of $R = 5$ days. Saturday and Sunday are not included. Empty containers are ordered every Tuesday morning and arrive to the supplier on Thursdays morning. Therefore, the first time step $r = 0$ corresponds to Tuesday and the ordering delay is $L_{ord} = 2$ days. The delivery delay is $L_{del} = 3$ as the full containers sent on Friday only arrive on Tuesday afternoon to the manufacturer. The partners would be interested in having a new container purchasing plan every year. Figure 9.1 presents the actual demands during the planning horizon. Figure 9.1a and represents the daily demand and Figure 9.1b presents the total demand between two arrival times. Even though a week is defined between two Tuesdays, we display the demand between two Thursdays because the demand between order arrivals gives more information on how many containers we need to order during the week.
A simple linear regression shows a clear increasing trend of the demand. Linear regression is a common statistical method to approximate a set of points with a line (or a hyperplane in a higher dimensional space). A description of linear regression can be found in [78]. We approximate the demand distributions with discretized normal distributions. We estimate the parameters of each distribution using a linear regression on the non-zero-demand values. Contrary to our container management problem with discrete demand,



(a) Daily demand.



(b) Demand between two order arrivals.

Figure 9.1: Demand data in our application. The red lines show a least-square linear regression of the non-zero-demands.

the linear regression approximation has an equation with real values, so we make another approximation to discretize the real-valued distribution from the linear regression.

We assume that the days with a zero-demand are known and that they represent special days. For instance, the week 11 in our data corresponds to week of Christmas. We have a set of $m = 158$ points (t_i, d_i), with $i \in [1, m]$, where t_i represents a day and d_i is the associated non-zero-demand value. We suppose that the expected demand is linearly increasing in i, hence is independent from the number of days with zero-demand between two consecutive points t_i and t_{i+1}.

The *least-squares method* is the most simple linear regression model. It assumes that the d_i are generated from a normal distribution $\mathcal{N}(a \cdot i + b, \sigma^2)$, where a and b are the linear regression parameters. In the least-squares method, the best values $\tilde{a}_{ls.d}$ and $\tilde{b}_{ls.d}$ of the parameters a and b minimizing the variance of the distribution. This is done by minimizing the sum of squared errors $\varepsilon_i := (d_i - b - a \cdot i)^2$:

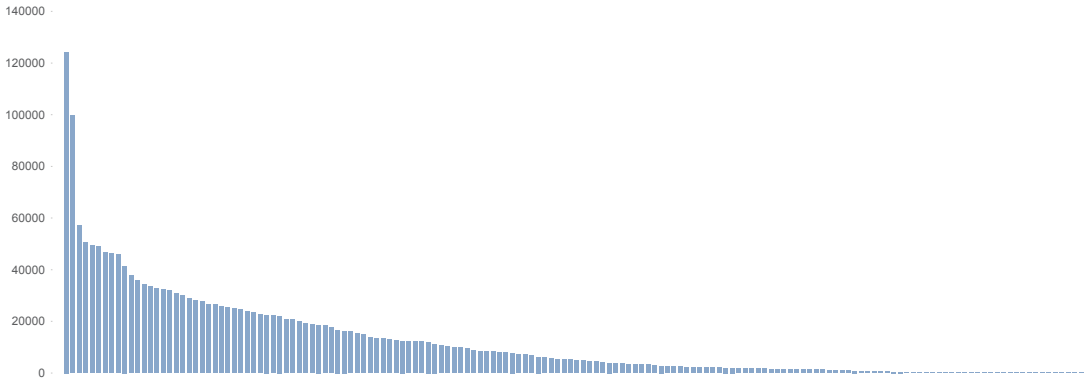
$$\tilde{a}_{ls.d},\ \tilde{b}_{ls.d} := arg \min_{a,b \in \mathbb{R}} \sum_{i=1}^{m} \varepsilon_i^2 \tag{9.1}$$

$$\forall i \in [1, m] : \ \varepsilon_i := d_i - b - a \cdot t_i \tag{9.2}$$

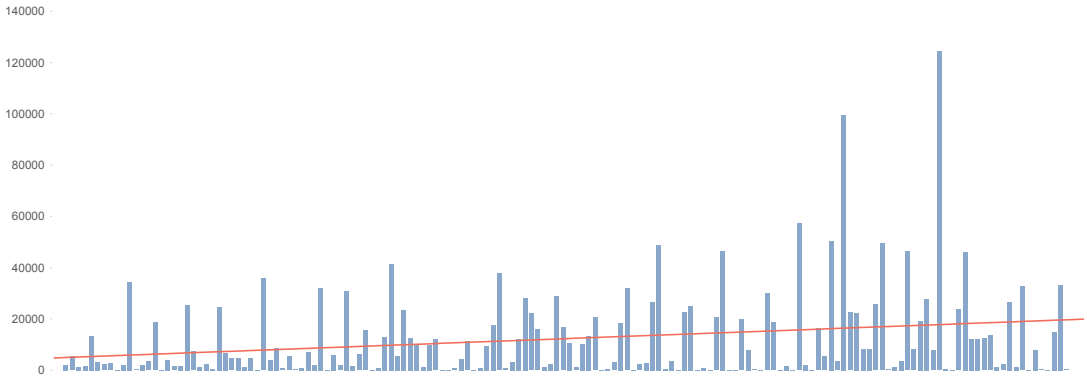
Figure 9.2 displays the residual squared errors $d_i = \tilde{a}_{ls.d} \cdot i + \tilde{b}_{ls.d}$ relative to the least-squares linear approximation, sorted by decreasing value in Figure 9.2a and by increasing time index in Figure 9.2b. In Figure 9.2a, we sort the errors by decreasing value and observe a hyperbole-like form. This goes along with our hypothesis that the error distribution is close to a normal distribution. However, Figure 9.2b highlights that the expected error increases over time. Therefore, the variance of the demand distribution should increases over time.
Therefore, we perform another linear regression over the residuals ε_i^2. For each i, we suppose that ε_i^2 follows a normal distribution with mean $\sigma_i^2 := \tilde{b}_{ls.v} + i \cdot \tilde{a}_{ls.v}$, where $\tilde{b}_{ls.v}$ and $\tilde{a}_{ls.v}$ are solution of the linear regression model:

$$\tilde{a}_{ls.v}, \ \tilde{b}_{ls.v} := arg \min_{a,b \in \mathbb{R}} \sum_{i=1}^{m} (\varepsilon_i^2 - b - a \cdot i)^2 \tag{9.3}$$

Since the variance is increasing, the error $d_i - \tilde{a}_{ls.d} \cdot i + \tilde{b}_{ls.d}$ of point with a small variance should be more important than the error of a point with



(a) Sorted by decreasing value.



(b) Sorted by increasing time.

Figure 9.2: Residual Errors $(d_i - \tilde{b}_{ls.d} - \tilde{a}_{ls.d} \cdot i)^2$ induced by the regression model $d_i = \tilde{a}_{ls.d} \cdot i + \tilde{b}_{ls.d}$.

high variance. However, we do not expend on other linear regression models associating weights to the points as it is not the central subject of this study. Thus, we stop the regression after a simple linear regression on the demand and on the variance of the demand. Using our linear regression results, we approximate the distribution of the i-th positive demand value with a normal distribution with mean $\mu_i := 167 + i \cdot 0.44$ and variance $\sigma_i^2 := 5100 + i \cdot 90$. Let $\mathbb{F}_{\mathcal{N},i}$ be the corresponding cumulative distribution function. We discretize every demand distribution $\mathcal{N}(\mu_i, \sigma_i^2)$ using a simple approximation. Fistly, we only consider positive integers in $[\mu_i - 2 \cdot \sigma_i, \mu_i + 2 \cdot \sigma_i]$. Secondly, every integral demand d in this interval is assigned to probability $\Omega \cdot \big(\mathbb{F}_{\mathcal{N},i}(d+0.5) - \mathbb{F}_{\mathcal{N},i}(d-0.5)\big)$, where $\Omega := \sum_d \big(\mathbb{F}_{\mathcal{N},i}(d+0.5) - \mathbb{F}_{\mathcal{N},i}(d-0.5)\big)$ is set so that the total probability equals 1. Therefore, the resulting function is a distribution and it is fully characterized by the mean μ_i and the variance σ_i^2 of the initial normal distribution. We call it a *discretized positive normal distribution*. Given a normal distribution $\mathcal{N}(\mu, \sigma^2)$, the interval $[\mu - 2 \cdot \sigma, \mu + 2 \cdot \sigma]$ represents 95% of the distribution density. Therefore, restricting the demand values to this interval is a good approximation.
In our simulations, we approximate the cumulative distribution function of a normal distribution with the formula of Bell [8]:

$$\forall x \in \mathbb{R},\ \mathbb{F}_{\mathcal{N},i}(x) := \frac{1}{2} \cdot \Big(1 + Sign(x) \cdot \sqrt{1 - \exp[-2 \cdot x^2/\pi]}\Big) \tag{9.4}$$

The formula is both simple and efficient, as its experimental maximum absolute error is of 0.003:

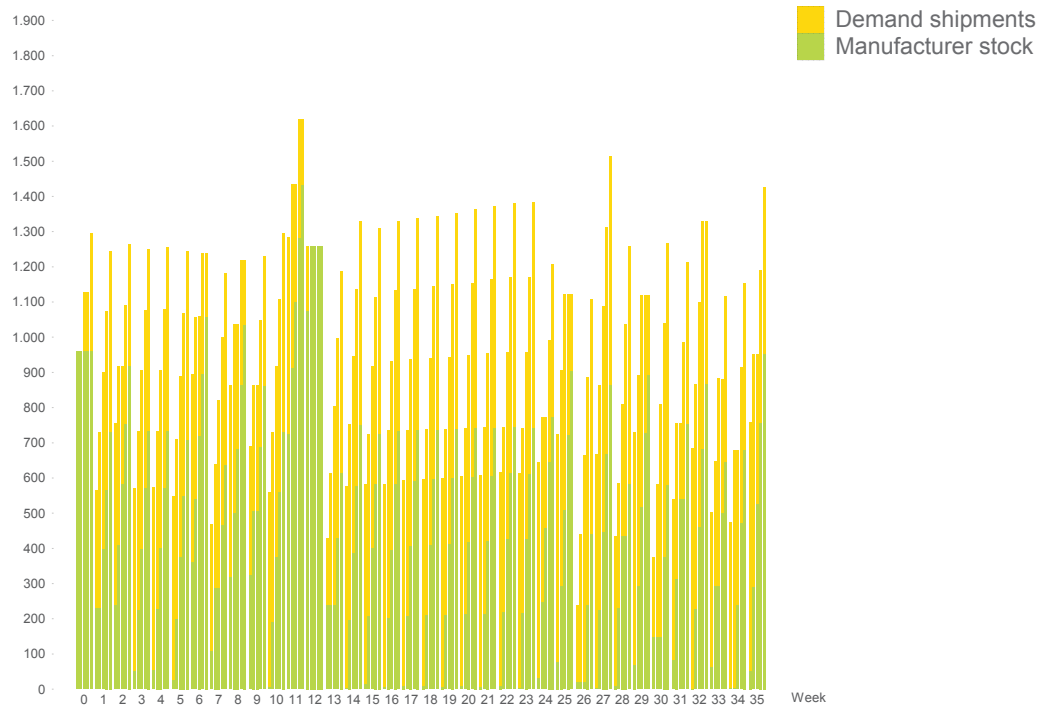
9.2 Description of the Experiments

In this chapter, we restrict our study to offline policies because an offline purchasing plan gives a better insight to the behavior of the system and the consequences of the purchasing decisions. In order to decrease the running time, we aggregate the demand in the simulations by packs of 30. This represents two pallets of fifteen containers. Nonetheless, the simulation results in the figures display quantities in containers.
We consider two stochastic offline policies, namely Algorithms $Mdp.F$ and $Sat.F$. We refer to Algorithm $Mdp.F$ as the *non-saturated policy* and to Algorithm $Sat.F$ as the saturated policy. Furthermore, we consider the deterministic algorithm $Flow$.5.2.
We use the demand regression model from Section 9.1. The starting demand distribution follows a discretized positive normal distribution with parameters $\mu_0 := 167$ and $\sigma_0^2 := 5100$. For every following positive demand, we increment the first parameter by 0.44 and the second one by 90.
We denote by $D_{t,r}$ the random variable of the demand distribution and by $D(t, r)$ the actual demand from our initial data. We can thus approximate

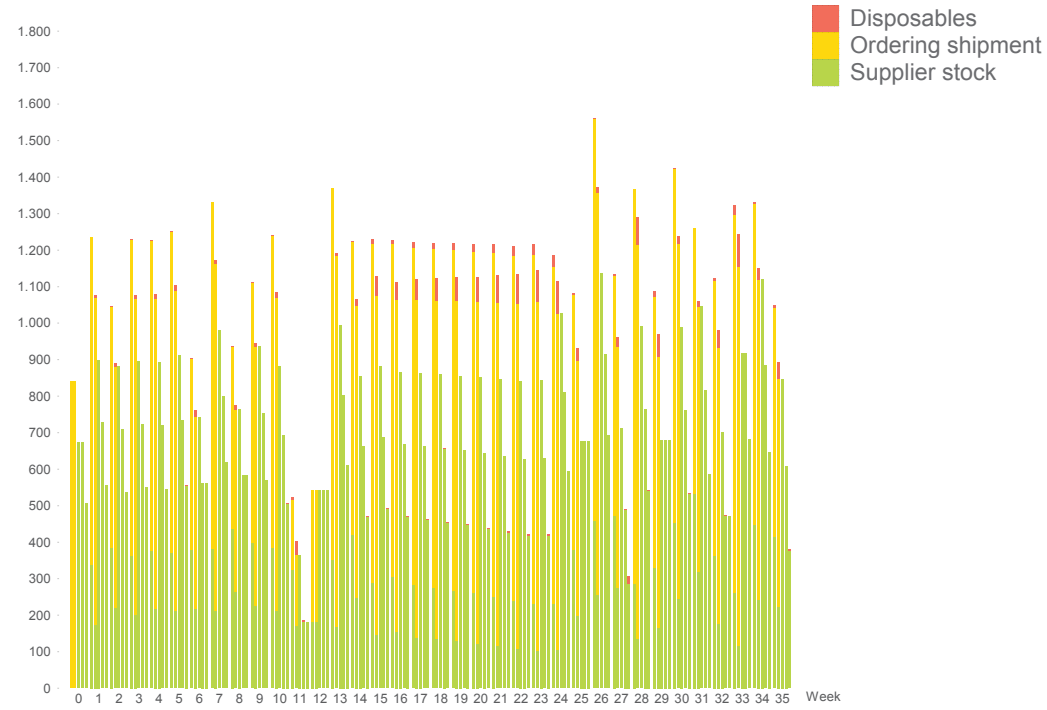
the distribution of demand $D_{t,r}$ by a discretized positive normal distribution with parameters $\mu_i := 167 + 0.44$ and $\sigma_i^2 := 5100 + i \cdot 90$ if $D(t,r)$ is the i-th positive demand of the time horizon. A zero-demand follows the zero distribution[1]. We refer to this demand model as the *stochastic demand model*.
Nevertheless, this demand model is quite complex so it may be difficult to understand the results. Thus, we first consider a simpler demand approximation such that $D_{t,r}$ follows a discretized positive normal distribution with parameters $\mu_{\text{tr}} := 167 + \text{tr} \cdot 0.44$ and $\sigma_{\text{tr}}^2 := 5100 + \text{tr} \cdot 90$, where $\text{tr} := t \cdot R + r = 5 \cdot t + r$. We refer to this demand model as the *stochastic positive demand model*. This second model enables us to quantify the influence of the days off on the process.
In addition, we use the deterministic algorithm to compute the best a posteriori purchasing plan using the actual demands. We call this model the *deterministic demand model*.

We now present our key performance indicators and explain how to read our figures. Figure 9.3 shows the expected evolution of the stocks and shipments over time for the non-saturated policy with the stochastic demand model. In both Figures 9.3a and 9.3b, the stock is represented in green and the containers arriving in future periods are in yellow. The expected number of disposables is in red. The x-axis is annotated with the week index and for each week we can see five bars corresponding to the stock and the shipments for each of the five working day of the week, namely Tuesday, Wednesday, Thursday, Friday and Monday. We let each week start with Tuesday as it corresponds to the ordering time. Note that the stock levels and shipment sizes are taken after the departure of the demand and before the arrival of the shipments. Therefore, the total supplier and manufacturer stock plus the total order and demand shipments is equal to the container fleet size in the system.
Firstly, we describe the process for the manufacturer, which can can be observed in Figure 9.3a. The manufacturer stock is given after the purchasing and ordering decision. The demand shipment at any time (t,r) is recorded after the departure of the shipment relative to demand $D_{t,r}$. Since the delivery time is $L_{del} = 3$, the demand shipments (in yellow) represent the total number of full containers which left the supplier over the past two days plus the shipment for the current day. At every day, a shipment arrives to the manufacturer whereas another departs from the supplier. Consequently, the total demand in shipments slowly increases over time, and this increase corresponds to the expected increase of demand over $L_{del} = 3$ days. The order size at week 11 is very small because it corresponds to the week of Christmas (see Figure 9.1b).
Secondly, Figure 9.3b presents the expected supplier stock, order size, and

[1] i.e. the demand is zero with probability 1.



(a) Manufacturer stock and total shipment of full containers.



(b) Supplier stock, order size, and number of disposables.

Figure 9.3: Expected evolution of the stock levels, shipments and number of disposables over time. The x-axis shows the index of each week, and for each week we can see the expected quantities for each of the five working days.

number of disposables bought at every day. The supplier stock is recorded after the demand departure of the day, so it corresponds to the stock at the beginning of the next day minus the order arrival. The order size does not change from Tuesday to Wednesday, so the ordering shipment has the same value in the first two days of each week. The order of empty containers arrives on Thursday, so the supplier stock increases every Thursday. At the week 11, the supplier stock and the order size are very low. The use of disposables can only be seen on Tuesdays and Wednesdays, i.e. between the order departure and arrival. Note that the expected supplier stock is always positive, even when the expected number of disposables is positive. The reason is that we are not looking at a single scenario with a demand realization, but rather at the expected stock and disposables over every possible demand scenario. Consequently, even though the supplier stock is empty whenever we buy disposables, the expected supplier stock is high because most of the time we do not need to buy disposables. We point out that the supplier stock on Wednesday is approximately equal one day of demand. Thus, if a shipment had a delay or was canceled, the supply chain would only have enough containers for one day. We would have to use disposables afterward. This remark opens up to the topic of robust optimization in a supply chain, which we have not studied in this thesis.
In our experiment results, we do not display the evolution of the process day by day, but week by week. We summary the information of Figure 9.3 as in Figure 9.4. We only present three variables, which are the purchase size, the expected number of idle containers and the expected number of disposables. The purchasing sizes are halved so that we can see the other quantities more clearly. We recall that the number of idle containers is equal to the manufacturer stock after ordering. The number of disposables for week t corresponds to the number of disposables bought from time (t, L_{ord}) to $(t+1, L_{ord}-1)$, i.e. between two ordering arrivals.
The expected number of disposables shows how many containers in the supply chain are unused. A good policy should make a compromise between having as few idle containers as possible, and having enough containers in the system to face a high demand. We explain the behavior of this result in the next section.

9.3 Experiments

9.3.1 The Stochastic Positive Demand Model

Firstly, we run a simulation for the stochastic positive demand model, with the saturated algorithm $Sat.F$ and the non-saturated algorithm $Mdp.F$. The simulation results are shown in Figure 9.5. The process starts without container, so every policy purchases a lot of containers at week $t = 0$.
Due to the presence of late demands, the manufacturer initially needs to

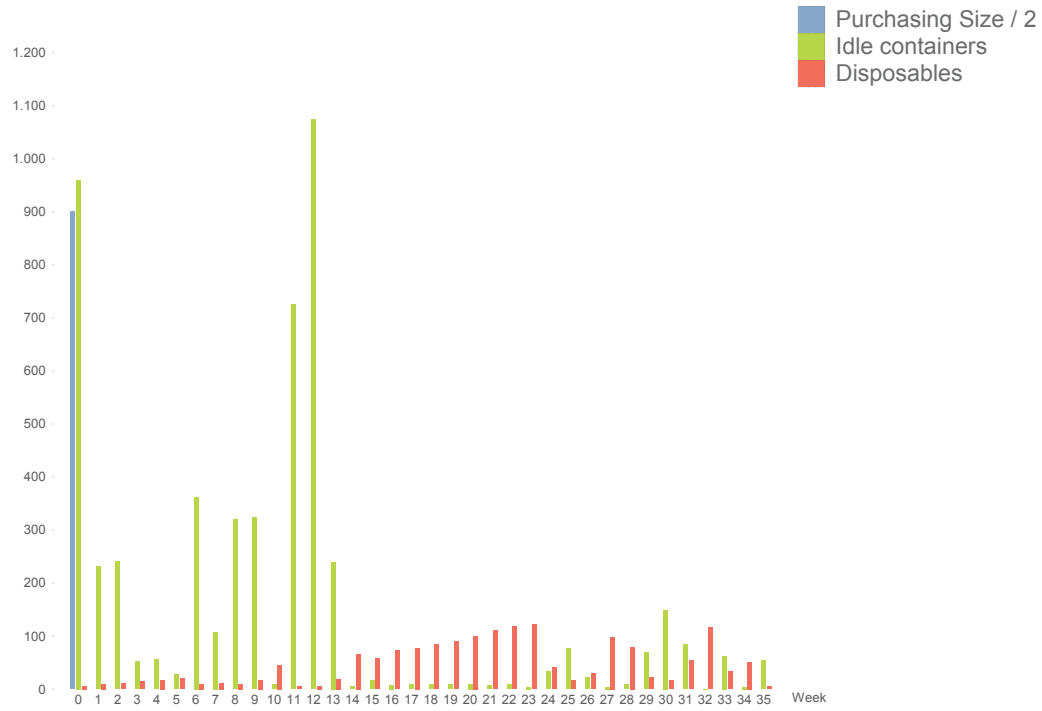


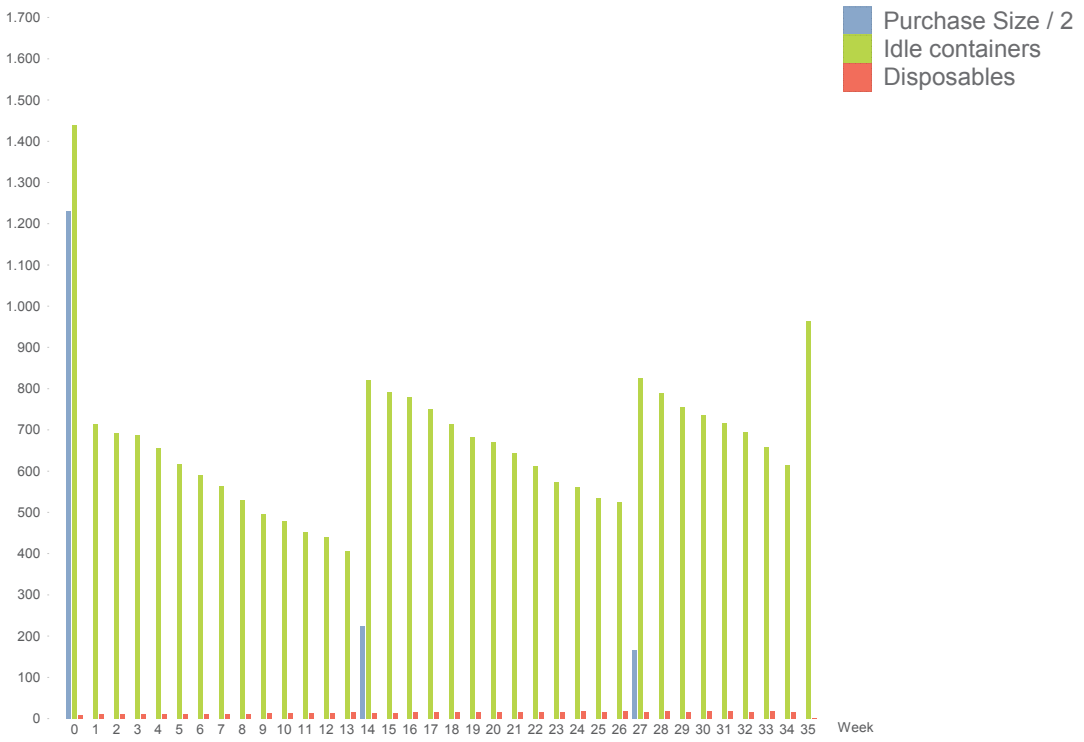
Figure 9.4: Expected number of disposables and idle containers for the saturated policy. The purchase size is halved for better visibility.

distribute the containers between the first two weeks. After the first two weeks, most containers used for a late demand at week t are ordered at week $t + 2$ and the decrease of idle containers corresponds to the expected demand increase. This explains why the number of idle containers is very high during the first week of the time horizon.

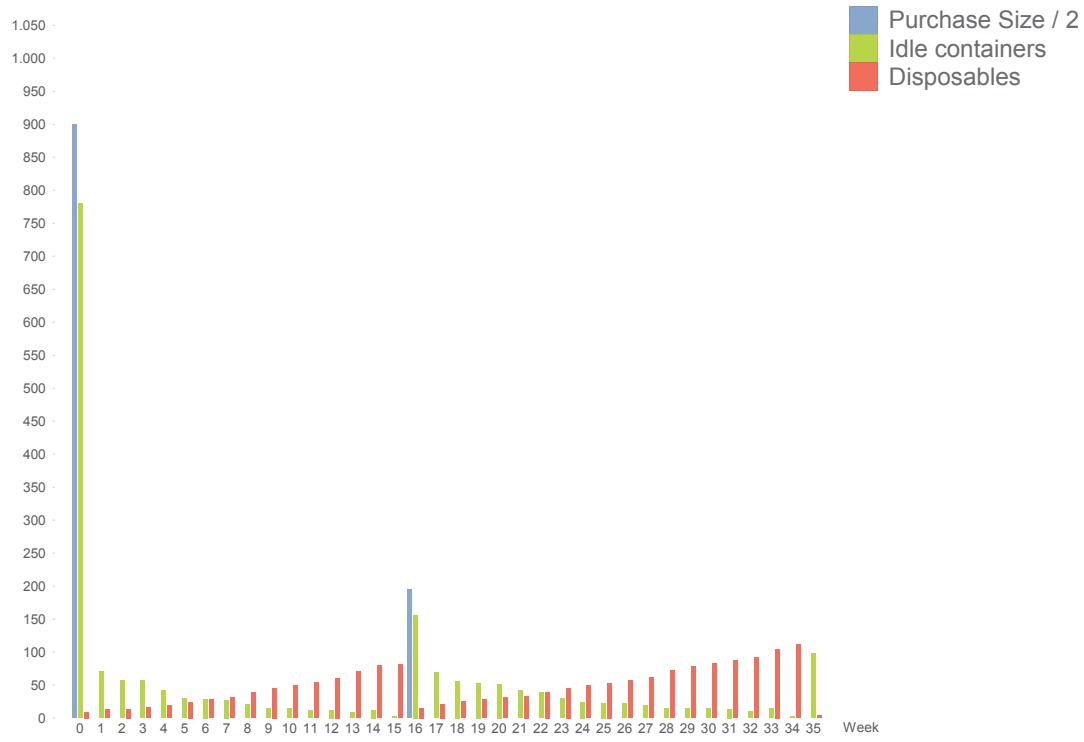
Moreover, at the last week of the time horizon, every policy will have many idle containers because we stop the demand on Monday, i.e. the last day before purchasing. Thus, the last week only represents the demand on Thursday, Friday, and Monday. This end of horizon effect is not relevant in our study but can be dealt with by adding two more days of demand.

This simulation highlights the usual behavior of both algorithms. On the one hand, the saturated policy purchases enough containers so that the supplier can always order the best order size. Because of that, very few disposables are necessary in the process for the whole time horizon. In return, the number of idle containers is very high and always exceed 20% of the whole fleet size. From a week to the next one, we can follow the expected number of idle containers decrease as a consequence of the increase of expected demand and demand variance. The policy purchases new containers approximately every three months.

On the other hand, the non-saturated policy induces a much lower cost by balancing the use of disposables and the container fleet size. This policy purchases less frequently containers and has a smaller fleet size. Further-



(a) Algorithm $Sat.F$, with cost 525 371.



(b) Algorithm $Mdp.F$, with cost 394 202.

Figure 9.5: Simulation results for the stochastic positive demand model.

more, after every purchasing, the expected number of disposables gradually increases as the container fleet size becomes too small for the demand. Thanks to these disposables, the number of idle containers is average less than 50 containers per week for the saturated policy, whereas it is between 400 and 800 for the saturated policy. Therefore, the non-saturated policy is wasting much fewer containers than the saturated policy.

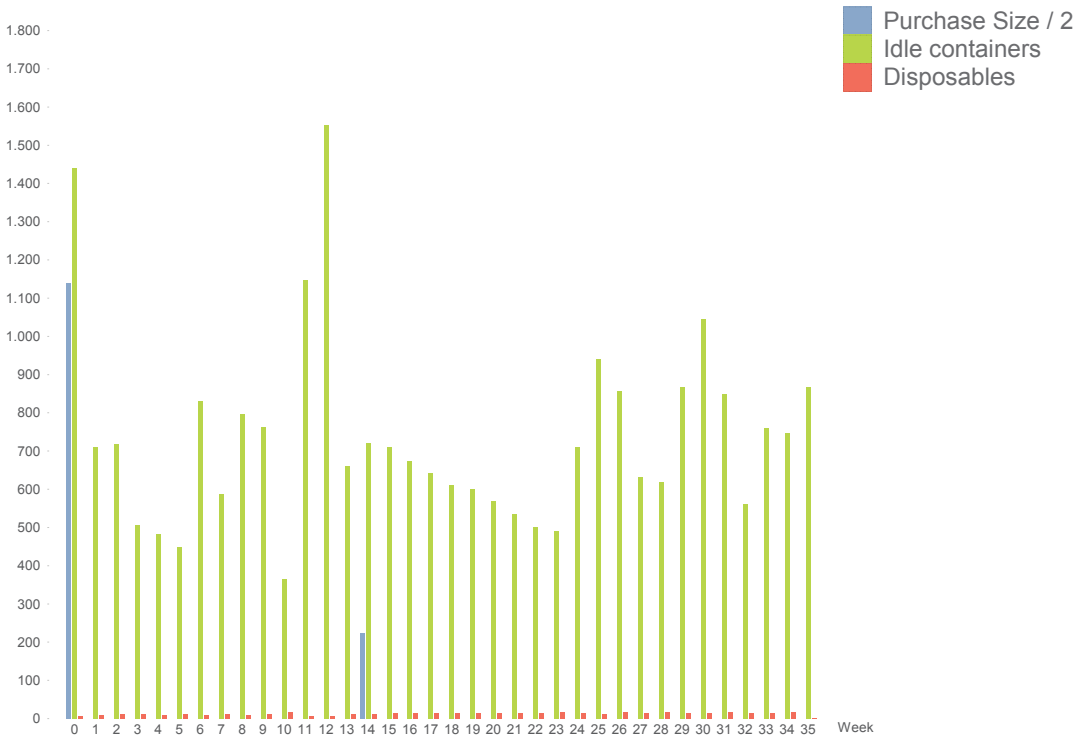
9.3.2 The Stochastic Demand Model

We now look at the results for the stochastic demand model, i.e. using the same days off and zero-demands as in the actual demand. The results for the saturated and non-saturated policy are presented in Figure 9.6.
To better understand the behavior of the process, we show in Figure 9.7 the number of zero-demands between two order arrivals. The number of zero-demands relative to week t is equal to the number of zero-demands between times (t, L_{ord}) and $(t+1, L_{ord}-1)$.
Firstly, both policies purchase less containers and have one less placement. In particular, Algorithm $Mdp.F$ only purchases containers at the beginning of the time horizon. If we look at the distribution of the days off, it appears that there is in average one zero-demand per week over the last third of the time horizon. Since the demand is increasing, it follows that we do not need many containers during the last weeks.
Furthermore, the number of idle containers and of disposables is fluctuating over time during the first and the last third of the time horizon. These fluctuations correspond to the number of zero-demands during the week. Namely, when there are several days off, the number of idle containers is higher and the number of bought disposables is lower. During the second third of the time horizon, there is no zero-demand. As a consequence, the process behavior in this time interval is similar to the stochastic positive demand model.

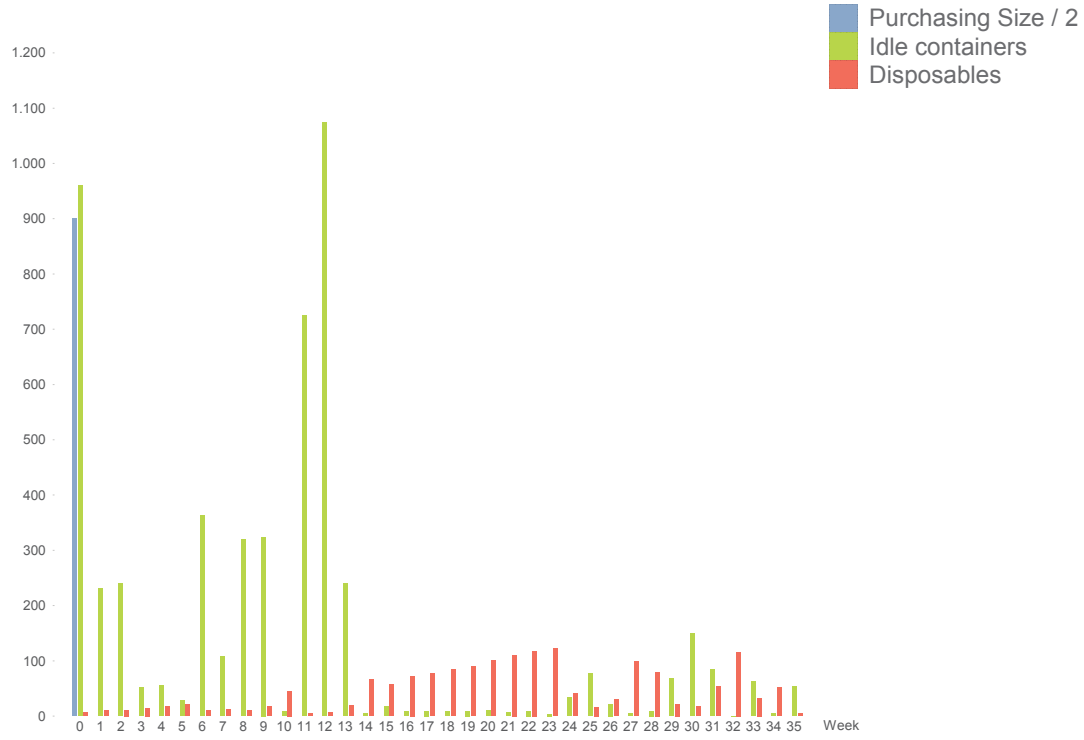
9.3.3 The Deterministic Demand Model

Finally, we run a simulation on the actual demand using the deterministic algorithm $Flow$.5.2. The results are presented in Figure 9.8.
A characteristic of deterministic experiments is that between every two placements[2] there is no idle container during at least one period.
Algorithm $Flow$.5.2 proposes a similar purchasing plan as the saturated policy under the stochastic demand model. However, it purchases less containers at the first period because a deterministic algorithm does not need to prepare for any worse case scenario. Under the stochastic demand model, the non-saturated policy has a rather small container fleet size, and would thus perform poorly under the actual demand scenario. Consequently, even

[2]and between the last placement and the last period of the time horizon



(a) Algorithm $Sat.F$, with cost 510 898.



(b) Algorithm $Mdp.F$, with cost 358 531.

Figure 9.6: Simulation results for the stochastic demand model.

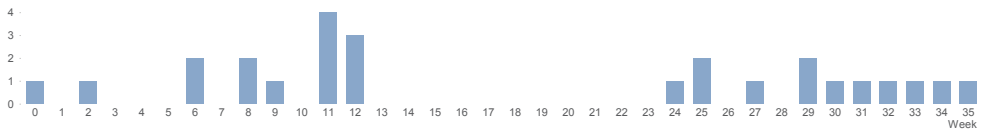


Figure 9.7: Number of zero-demands between two consecutive order arrivals.

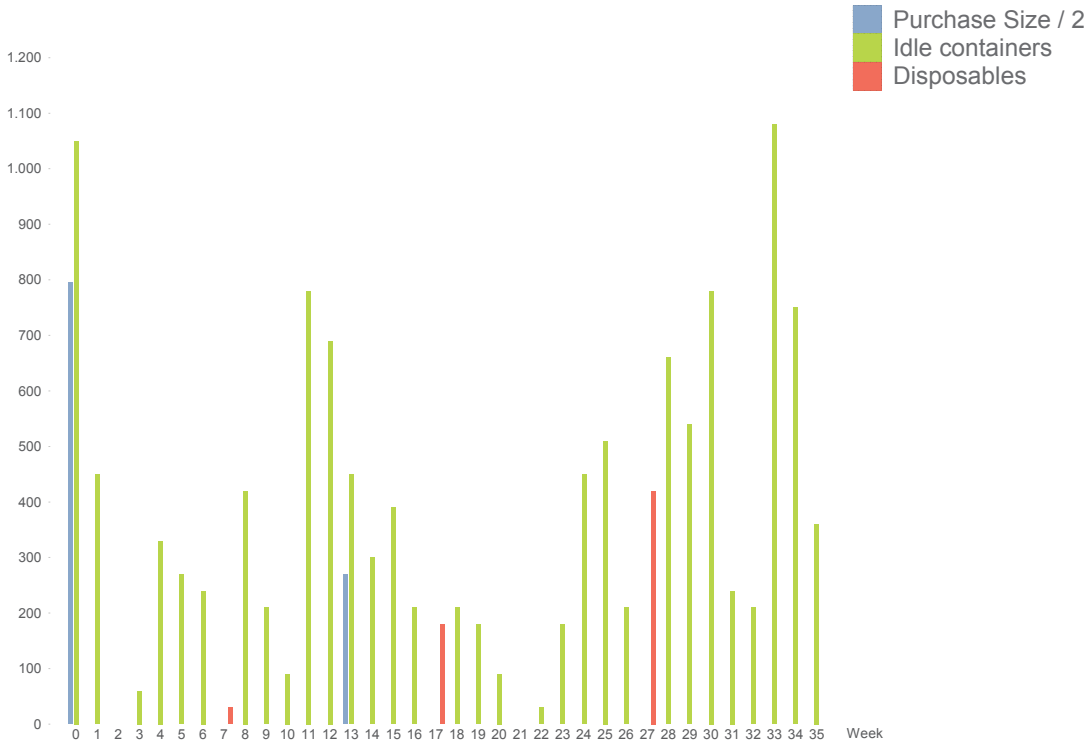


Figure 9.8: Simulation results of Algorithm *Flow*.5.2 for the deterministic demand model, with cost 322 770.

though the non-saturated algorithm provides a very good policy in average, we should forecast the future demand as precisely as possible. Note that the first demand of the time horizon has an mean of 167 and a standard deviation of $\sqrt{5100} \approx 71$. Therefore, there is a huge variability in the possible values of the demand, which is to be expected from the fluctuations of the actual demands (see Figure 9.1a).

Chapter 10

Conclusion and Future Research

The purpose of this thesis is to optimize container management in closed-loop supply chains with increasing demand. The decisions to take are not only the operational task of repositioning empty containers but also the tactical choice of purchasing new containers. In this chapter we review our findings and open up to future research directions.

10.1 Problem Statement and Contributions

In our logistic model, we assume that the supply chain partners have already agreed to have the manufacturer send empty containers to its supplier on a periodic basis.
We make three major assumptions. Firstly, the manufacturer pays a setup cost whenever he purchases new containers. Secondly, there is an ordering delay and a delivery delay whenever containers are sent between the two locations, which has a big impact on the optimal container fleet size. Nonetheless, the total delay is assumed shorter than the ordering period. Thirdly, the demands do not need to be entirely fulfilled with containers as the supplier can buy expensive one-way disposables as substitute. This situation corresponds to a lost-sales scenario.
These assumptions make the modeling more realistic but also much more complex to solve. Removing the setup cost will likely make the manufacturer purchase new containers at every period. This will hence significantly change the structure of the solution. Moreover, as shown in Chapter 6, a policy neglecting the lead times purchases much less containers than a policy taking them into account. A policy neglecting the lead time generates performs poorly when applied to the problem with positive lead times. Furthermore, we experimentally showed in Chapters 6 and 7 that allowing disposables greatly decreases the policy cost.

In the literature, very few papers consider a container management problem with container purchasing. Our problem lies between two big streams of research: lot-sizing problems and container management problems. Lot-sizing problems consider the purchasing of items in a supply chain at a setup cost, whereas container management problems usually study repositioning strategies of empty containers between equally important locations. We consider the purchasing and the repositioning of empty containers in a closed-loop supply chain where the two actors, i.e. the manufacturer and its supplier, have a different role. Our questionings are in a context of reverse logistics. We assume that the partners of the supply chain have already agreed on the structure of the reverse flow of empty containers.
Many researchers study a lot-sizing problem with remanufacturing where the remanufacturing is very close to the return of containers. However, these models neglect the clear relationship between the number of sold items and the number of returning items. This thesis is one of the first attempts at developing a lot-sizing model where the items are containers which always return to the manufacturer after use.

10.2 Resolution Methodology

We study the deterministic and the stochastic versions of the problem. We solve both versions by adapting the Wagner-Within algorithm. For every pair of periods we compute a good purchase size and ordering policy between these periods, assuming that they are consecutive placements. We build our final policy from these *partial policies* using a dynamic program. In the literature, the Wagner-Within framework has been successfully applied to lot-sizing problems.
The deterministic problem ($DCPP$) is modeled as a minimum cost flow on a network with fixed-plus-linear cost. Each of the partial policies is modeled as a minimum linear-cost flow. Without either setup cost, disposables or lead times, $DCPP$ can be solved in polynomial time. However, under the three problem assumptions, it is very complex to compute an optimal solution and we conjecture the problem to be NP-hard.
We consider the special case where the demand is steadily increasing in Chapter 4 and any demand pattern in Chapter 5. These algorithms are experimentally shown to be fast and to produce near-optimal policies. In Chapter 6, we improve our the computational speed of our algorithm by making use of the network properties.
More precisely, we suppose in Chapter 4 that there is no need to buy disposables shortly after a placement. We create partial policies which are independent from each others and deduce an optimal solution in polynomial time. In Chapter 5, we propose alternative algorithms and give an optimality certificate. The algorithm is proven to be optimal if there is exists an

optimal policy letting at least one container idle at each purchasing period. Furthermore, we develop a more general framework to compute very good policies but with a high time complexity.

Under stochastic demands, we model each partial policy as a Markov decision process during a time window. Each partial policy is computed in a backward sequence, starting from an approximation of the cost of every state of the system at the end of the time window.
From a practitioner point of view, there is no clear way of defining what is the best policy when the demand is stochastic, in contrast to the deterministic setting. We generalize four different strategies from the literature. All of them take the ordering decision in an online manner, as late as possible. The online strategy takes each purchasing decision as late as possible. Thus, an optimal online policy minimizes the cost over every solution to the problem. On the opposite, the offline strategy takes the purchasing decision right from the start. Compared to the online strategy, the offline strategy is not using the demand realization and the state of the process at any purchasing time besides the first one. We propose two compromises. The quasi-offline strategy, which has been presented with the two above by Bookbinder and Tan [11], decides on the purchasing times at the start whereas the actual quantities are decided as late as possible. The quasi-online strategy generalizes in particular the rolling-horizon heuristic used by Silver and Meal (see [106]). It decides at each placement on the purchasing size and the timing of the next placement. Our online and quasi-online algorithms compute the optimal policy of their respective strategies. However, our offline and quasi-offline algorithms make further approximations while using the same framework as the quasi-online policy.

10.3 Future Research

This thesis focuses on a container purchasing problem in a simple 2-echelon supply chain with a single supplier and a single manufacturer.
In the deterministic case, the extension to several suppliers is straightforward, as long as the total lead time between the manufacturer and each supplier is shorter than one period length. An extension to longer lead times follows with a higher complexity if the minimum distance between two placement is shorter than the total ordering and delivery delays. However, many problems arise when we consider several manufacturer locations. In particular, we must decide on either a joint or a separate setup cost over every manufacturer locations, or else if only a single location allowed to purchase new containers. Moreover, extending our algorithm appears challenging, because we need to find out a good repartition of the containers between the manufacturer locations at purchasing time. Furthermore, the NP-hardness of the deterministic problem is still an open question.

For the stochastic problem, we only solved the problem optimally for two of the four strategies. While the quasi-offline heuristic seems nearly optimal in every scenario, this is not the case for the offline heuristic. Our algorithm performs very well when the distance between consecutive placements is long enough so that the number of containers we should purchase does not depend on the number of containers we need for the first period following the placement. Otherwise, the reference state we used to approximate the optimal purchase quantity underestimates the required fleet size because it forgets the containers we need to have in stock and in transports at purchasing. We proposed a second reference state using the optimal saturated order policy to replace it. Nevertheless, it would be interesting to either find a way to guess what is a good reference state for each cost structure. an alternative to reference states keeping the same algorithmic structure
Moreover, a Markov decision process can be easily extended to several suppliers and manufacturer, but this is at the cost of a huge increase in time complexity. This makes our algorithms unpractical. The best way to solve this curse of dimensionality would be to use approximate dynamic programming, which we presented in chapter 8. As hinted in the literature [90], we need a good knowledge on the solution structure if we want to use approximate dynamic programming to its full potential. In this thesis, we showed the $L^\natural$-convexity of the cost functions, which is a good start to analyze the problem properties.
Another meaningful study would be to develop a container purchasing problem with demand forecasts. In particular, some companies consider a rolling horizon using forecasts with different degrees of precision. For instance, we may have an estimation of the daily demand for the coming two months, then a estimation of the weekly demand for the following four months, and an estimation of the monthly demand for the last six months of the one year time-horizon.
Finally, further research is needed to find a simple formula approximating the best ordering size, the best purchasing size and the best next period for placement. In particular, a dual-balancing policy or a heuristic as presented in Chapter 8 would greatly help practitioners.

Bibliography

[1] A. Aggarwal and J.K. Park. Improved algorithms for economic lot size problems. *Operations research*, 41(3):549–571, 1993.

[2] RK. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc. Upper Saddle River, New Jersey, 1993.

[3] D. Aksen, K. Altınkemer, and S. Chand. The single-item lot-sizing problem with immediate lost sales. *European Journal of Operational Research*, 147(3):558–566, 2003.

[4] M.A. Aloulou, A. Dolgui, and M.Y. Kovalyov. A bibliography of non-deterministic lot-sizing models. *International Journal of Production Research*, 52(8):2293–2310, 2014.

[5] Joachim Arts, Retsef Levi, Geert-Jan van Houtum, and Bert Zwart. Base-stock policies for lost-sales models: Aggregation and asymptotics. 2015.

[6] R.G. Askin. A procedure for production lot sizing with probabilistic dynamic demand. *Aiie Transactions*, 13(2):132–137, 1981.

[7] I. Barany, T. Van Roy, and L.A. Wolsey. *Uncapacitated lot-sizing: The convex hull of solutions*. Springer, 1984.

[8] Jeff Bell. A simple and pragmatic approximation to the normal cumulative probability distribution. *Available at SSRN 2579686*, 2015.

[9] Richard Bellman. The theory of dynamic programming. Technical report, DTIC Document, 1954.

[10] Marco Bijvank and Iris FA Vis. Lost-sales inventory theory: A review. *European Journal of Operational Research*, 215(1):1–13, 2011.

[11] J.H. Bookbinder and J-Y. Tan. Strategies for the probabilistic lot-sizing problem with service-level constraints. *Management Science*, 34(9):1096–1108, 1988.

[12] A. Chandoul, V-D. Cung, and F. Mangione. Reusable containers within reverse logistic context. 2007.

[13] A. Chandoul, V-D. Cung, and F. Mangione. Optimal repositioning and purchasing policies in returnable container management. In *Industrial Engineering and Engineering Management, 2009. IEEM 2009. IEEE International Conference on*, pages 1439–1443. IEEE, 2009.

[14] W. Chen, M. Dawande, and G. Janakiraman. Fixed-dimensional stochastic dynamic programs: An approximation scheme and an inventory application. *Operations Research*, 62(1):81–103, 2014.

[15] R.K. Cheung and C-Y. Chen. A two-stage stochastic network model and solution methods for the dynamic empty container allocation problem. *Transportation science*, 32(2):142–162, 1998.

[16] Chi Chiang. Optimal ordering policies for periodic-review systems with replenishment cycles. *European Journal of Operational Research*, 170(1):44–56, 2006.

[17] A. Cimino, R. Diaz, F. Longo, and G. Mirabelli. Empty containers repositioning: A state of the art overview. In *Proceedings of the 2010 Spring Simulation Multiconference*, page 72. Society for Computer Simulation International, 2010.

[18] T.G. Crainic, M. Gendreau, and P. Dejax. Dynamic and stochastic models for the allocation of empty containers. *Operations research*, 41(1):102–126, 1993.

[19] P.J. Dejax and T.G. Crainic. Survey paper-a review of empty flows and fleet management models in freight transportation. *Transportation Science*, 21(4):227–248, 1987.

[20] M. Di Francesco. New optimization models for empty container management. 2007.

[21] W.A. Donaldson. Inventory replenishment policy for a linear trend in demand–an analytical solution. *Operational Research Quarterly*, pages 663–670, 1977.

[22] J-X. Dong and D-P. Song. Container fleet sizing and empty repositioning in liner shipping systems. *Transportation Research Part E: Logistics and Transportation Review*, 45(6):860–877, 2009.

[23] Jing-Xin Dong and Dong-Ping Song. Quantifying the impact of inland transport times on container fleet sizing in liner shipping services with uncertainties. *OR spectrum*, 34(1):155–180, 2012.

[24] A.L. Erera, J.C. Morales, and M. Savelsbergh. Robust optimization for empty repositioning problems. *Operations Research*, 57(2):468–483, 2009.

[25] A.L. Erera, J.C. Morales, and Martin Savelsbergh. Global intermodal tank container management for the chemical industry. *Transportation Research Part E: Logistics and Transportation Review*, 41(6):551–566, 2005.

[26] R.E. Erickson, C.L. Monma, and A.F. Veinott Jr. Send-and-split method for minimum-concave-cost network flows. *Mathematics of Operations Research*, 12(4):634–664, 1987.

[27] A. Federgruen and M. Tzur. A simple forward algorithm to solve general dynamic lot sizing models with n periods in 0 (n log n) or 0 (n) time. *Management Science*, 37(8):909–925, 1991.

[28] G. Gao. An operational approach for container control in liner shipping. *Logistics and Transportation Review*, 30(3):267–282, 1994.

[29] M.R. Garey and D.S. Johnson. Computers and intractability: a guide to the theory of np-completeness. 1979. *San Francisco, LA: Freeman*, 1979.

[30] C.H. Glock and T. Kim. Container management in a single-vendor-multiple-buyer supply chain. *Logistics Research*, 7(1):1–16, 2014.

[31] B. Golany, J. Yang, and G. Yu. Economic lot-sizing with remanufacturing options. *Iie Transactions*, 33(11):995–1003, 2001.

[32] David A Goldberg, Dmitriy A Katz-Rogozhnikov, Yingdong Lu, Mayank Sharma, and Mark S Squillante. Asymptotic optimality of constant-order policies for lost sales inventory models with large lead times. *Mathematics of Operations Research*, 2016.

[33] S.C. Graves and J.B. Orlin. A minimum concave-cost dynamic network flow problem with an application to lot-sizing. *Networks*, 15(1):59–71, 1985.

[34] Tore Grünert and Stefan Irnich. *Optimierung im Transport*. Shaker Verlag, 2005.

[35] Y-P. Guan, S. Ahmed, G.L. Nemhauser, and A.J. Miller. A branch-and-cut algorithm for the stochastic uncapacitated lot-sizing problem. *Mathematical Programming*, 105(1):55–84, 2006.

[36] Y-P. Guan and A.J. Miller. Polynomial-time algorithms for stochastic uncapacitated lot-sizing problems. *Operations Research*, 56(5):1172–1183, 2008.

[37] G.M. Guisewite and P.M. Pardalos. Minimum concave-cost network flow problems: Applications, complexity, and algorithms. *Annals of Operations Research*, 25(1):75–99, 1990.

[38] G.M. Guisewite and P.M. Pardalos. A polynomial time solvable concave network flow problem. *Networks*, 23(2):143–147, 1993.

[39] N. Halman, D. Klabjan, C-L. Li, J. Orlin, and D. Simchi-Levi. Fully polynomial time approximation schemes for stochastic dynamic programs. *SIAM Journal on Discrete Mathematics*, 28(4):1725–1796, 2014.

[40] N. Halman, D. Klabjan, M. Mostagir, J. Orlin, and D. Simchi-Levi. A fully polynomial-time approximation scheme for single-item stochastic inventory control with discrete demand. *Mathematics of Operations Research*, 34(3):674–685, 2009.

[41] F.W. Harris. How many parts to make at once. 1913.

[42] K.K. Haugen, A. Løkketangen, and D.L. Woodruff. Progressive hedging as a meta-heuristic applied to stochastic lot-sizing. *European Journal of Operational Research*, 132(1):116–122, 2001.

[43] Q. He. Topics in discrete optimization: models, complexity and algorithms. 2013.

[44] Q. He, S. Ahmed, and G.L. Nemhauser. Minimum concave cost flow over a grid network. *Mathematical Programming*, 150(1):79–98, 2012.

[45] David C Heath and Peter L Jackson. Modeling the evolution of demand forecasts ith application to safety stock analysis in production/distribution systems. *IIE transactions*, 26(3):17–30, 1994.

[46] M.J.R. Helmrich, R. Jans, W.J. van den Heuvel, and A.P.M. Wagelmans. Economic lot-sizing with remanufacturing: complexity and efficient formulations. Technical report, Econometric Institute Research Papers, 2010.

[47] W.J. Hopp. Ten most influential papers of management science's first fifty years. *Management Science*, 50(12_supplement):1763–1763, 2004.

[48] Woonghee Tim Huh and Ganesh Janakiraman. On the optimal policy structure in serial inventory systems with lost sales. *Operations Research*, 58(2):486–491, 2010.

[49] Woonghee Tim Huh, Ganesh Janakiraman, John A Muckstadt, and Paat Rusmevichientong. An adaptive algorithm for finding the optimal base-stock policy in lost sales inventory systems with censored demand. *Mathematics of Operations Research*, 34(2):397–416, 2009.

[50] Woonghee Tim Huh, Ganesh Janakiraman, John A Muckstadt, and Paat Rusmevichientong. Asymptotic optimality of order-up-to policies in lost sales inventory systems. *Management Science*, 55(3):404–420, 2009.

[51] G. Hurley, P. Jackson, R. Levi, R.O. Roundy, and D.B. Shmoys. New policies for stochastic inventory control models–theoretical and computational results. Technical report, Citeseer, 2007.

[52] A. Imai, K. Shintani, and S. Papadimitriou. Multi-port vs. hub-and-spoke port calls by containerships. *Transportation Research Part E: Logistics and Transportation Review*, 45(5):740–757, 2009.

[53] N. Jami and M. Schröder. Tactical and operational models for the management of a warehouse. In *Dynamics in Logistics*, pages 655–665. Springer, 2016.

[54] N. Jami, M. Schroeder, and K-H. Küfer. A model and polynomial algorithm for purchasing and repositioning containers. *Management and Control of Production and Logistics*, 7, 2016.

[55] N. Jami, M. Schroeder, and K-H. Küfer. Online and offline container purchasing and repositioning problem. *International Conference on Computational Logistics*, 7:159–174, 2016.

[56] G. Janakiraman and J.A. Muckstadt. Periodic review inventory control with lost sales and fractional lead times. *School of Operations Research and Industrial Engineering, Cornell University*, 2004.

[57] Ganesh Janakiraman, Sridhar Seshadri, and J George Shanthikumar. A comparison of the optimal costs of two canonical inventory systems. *Operations Research*, 55(5):866–875, 2007.

[58] Søren Glud Johansen and Anders Thorstenson. Pure and restricted base-stock policies for the lost-sales inventory system with periodic review and constant lead times. In *15th International Symposium on Inventories*, 2008.

[59] David S Johnson. The np-completeness column: an ongoing guide. *Journal of algorithms*, 13(3):502–524, 1992.

[60] H. Jula, M. Dessouky, P. Ioannou, and A. Chassiakos. Container movement by trucks in metropolitan networks: modeling and optimization. *Transportation Research Part E: Logistics and Transportation Review*, 41(3):235–259, 2005.

[61] IA. Karimi, M. Sharafali, and H. Mahalingam. Scheduling tank container movements for chemical logistics. *AIChE journal*, 51(1):178–197, 2005.

[62] S. Karlin and H. Scarf. Inventory models of the arrow-harris-marschak type with time lag. *Studies in the mathematical theory of inventory and production*, (1):155, 1958.

[63] G.P. Kiesmüller and E.A. Van der Laan. An inventory model with dependent product demands and returns. *International journal of production economics*, 72(1):73–87, 2001.

[64] A.J. Kleywegt, A. Shapiro, and T. Homem-de Mello. The sample average approximation method for stochastic discrete optimization. *SIAM Journal on Optimization*, 12(2):479–502, 2002.

[65] L. Kroon and G. Vrijens. Returnable containers: an example of reverse logistics. *International Journal of Physical Distribution & Logistics Management*, 25(2):56–68, 1995.

[66] S-W. Lam, L-H. Lee, and L-C. Tang. An approximate dynamic programming approach for the empty container allocation problem. *Transportation Research Part C: Emerging Technologies*, 15(4):265–277, 2007.

[67] C-Y. Lee, S. Çetinkaya, and W. Jaruphongsa. A dynamic model for inventory lot sizing and outbound shipment scheduling at a third-party warehouse. *Operations Research*, 51(5):735–747, 2003.

[68] L-H. Lee, E-P. Chew, and Y. Luo. Inventory-based empty container repositioning in a multi-port system. In *Proceedings of the First International Conference on Advanced Communications and Computation (INFOCOMP 2011)*, pages 23–29, 2011.

[69] R. Levi, G. Janakiraman, and M. Nagarajan. A 2-approximation algorithm for stochastic inventory control models with lost sales. *Mathematics of Operations Research*, 33(2):351–374, 2008.

[70] R. Levi, M. Pál, R.O. Roundy, and D.B. Shmoys. Approximation algorithms for stochastic inventory control models. *Mathematics of Operations Research*, 32(2):284–302, 2007.

[71] R. Levi, R.O. Roundy, D.B. Shmoys, and V-A. Truong. Approximation algorithms for capacitated stochastic inventory control models. *Operations Research*, 56(5):1184–1199, 2008.

[72] C-B. Li, F. Liu, H-J. Cao, and Q-L. Wang. A stochastic dynamic programming based model for uncertain production planning of remanufacturing system. *International Journal of Production Research*, 47(13):3657–3668, 2009.

[73] J-A. Li, S.C.H. Leung, Y. Wu, and K. Liu. Allocation of empty containers between multi-ports. *European Journal of Operational Research*, 182(1):400–412, 2007.

[74] J-A. Li, K. Liu, S.C.H. Leung, and K.K. Lai. Empty container management in a port with long-run average criterion. *Mathematical and Computer Modelling*, 40(1):85–100, 2004.

[75] S. Meisel. *Anticipatory optimization for dynamic decision making*, volume 51. Springer Science & Business Media, 2011.

[76] R. A Melo and L.A. Wolsey. Uncapacitated two-level lot-sizing. *Operations Research Letters*, 38(4):241–245, 2010.

[77] R.A. Melo and L.A. Wolsey. Mip formulations and heuristics for two-level production-transportation problems. *Computers & Operations Research*, 39(11):2776–2786, 2012.

[78] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*. John Wiley & Sons, 2015.

[79] I-K. Moon, A-D. Do Ngoc, and Y-S. Hur. Positioning empty containers among multiple ports with leasing and purchasing considerations. *OR spectrum*, 32(3):765–786, 2010.

[80] T.E. Morton. Bounds on the solution of the lagged optimal inventory equation with no demand backlogging and proportional costs. *SIAM review*, 11(4):572–596, 1969.

[81] T.E. Morton. The near-myopic nature of the lagged-proportional-cost inventory problem with lost sales. *Operations Research*, 19(7):1708–1716, 1971.

[82] John A Muckstadt and Amar Sapra. *Principles of inventory management: When you are down to four, order more*. Springer Science & Business Media, 2010.

[83] K. Murota. *Discrete convex analysis*. SIAM, 2003.

[84] A. Olivo, P. Zuddas, M. Di Francesco, and A. Manca. An operational model for empty container management. *Maritime Economics & Logistics*, 7(3):199–222, 2005.

[85] J.B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations research*, 41(2):338–350, 1993.

[86] U. Özen, M.K. Doğru, and S.A. Tarim. Static-dynamic uncertainty strategy for a single-item stochastic inventory control problem. *Omega*, 40(3):348–357, 2012.

[87] G.S. Piperagkas, I. Konstantaras, K. Skouri, and K.E. Parsopoulos. Solving the stochastic dynamic lot-sizing problem through nature-inspired heuristics. *Computers & Operations Research*, 39(7):1555–1565, 2012.

[88] Y. Pochet and L.A. Wolsey. *Production planning by mixed integer programming*. Springer Science & Business Media, 2006.

[89] Warren B Powell. What you should know about approximate dynamic programming. *Naval Research Logistics (NRL)*, 56(3):239–249, 2009.

[90] Warren B Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, volume 842. John Wiley & Sons, 2011.

[91] Warren B Powell and Huseyin Topaloglu. Stochastic programming in transportation and logistics. *Handbooks in operations research and management science*, 10:555–635, 2003.

[92] I.N. Pujawan. The effect of lot sizing rules on order variability. *European Journal of Operational Research*, 159(3):617–635, 2004.

[93] I.N. Pujawan and E.A. Silver. Augmenting the lot sizing order quantity when demand is probabilistic. *European Journal of Operational Research*, 188(3):705–722, 2008.

[94] K. Richter and M. Sombrutzki. Remanufacturing planning for the reverse wagner/whitin models. *European Journal of Operational Research*, 121(2):304–315, 2000.

[95] K. Richter and J. Weber. The reverse wagner/whitin model with variable manufacturing and remanufacturing cost. *International Journal of Production Economics*, 71(1):447–456, 2001.

[96] R. Rossi, O.A. Kilic, and S.A. Tarim. A unified modeling approach for the static-dynamic uncertainty strategy in stochastic lot-sizing. *arXiv preprint arXiv:1307.5942*, 2013.

[97] R. Rossi, S.A. Tarim, B. Hnich, and S. Prestwich. Replenishment planning for stochastic inventory systems with shortage cost. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 229–243. Springer, 2007.

[98] R. Rossi, S.A. Tarim, B. Hnich, and S. Prestwich. Computing replenishment cycle policy under non-stationary stochastic lead time. *International Journal of Production Economics. v127 i1*, pages 180–189, 2008.

[99] R. Rossi, S.A. Tarim, B. Hnich, and S. Prestwich. A state space augmentation algorithm for the replenishment cycle inventory policy. *International Journal of Production Economics*, 133(1):377–384, 2011.

[100] R. Rossi, S.A. Tarim, B. Hnich, and S. Prestwich. Constraint programming for stochastic inventory systems under shortage cost. *Annals of Operations Research*, 195(1):49–71, 2012.

[101] M. Schroeder, N. Jami, U. Beissert, and M. Motta. Konzeptionierung eines integrierten modellbasierten ansatzes zur prognose von transportlogistischen und intralogistischen ereignissen in logistiknetzwerken. *Simulation in Production and Logistics 2015*, 16(157):147–156, 2015.

[102] T. Schulz. A new silver–meal based heuristic for the single-item dynamic lot sizing problem with returns and remanufacturing. *International Journal of Production Research*, 49(9):2519–2533, 2011.

[103] WS. Shen and CM. Khoong. A dss for empty container distribution planning. *Decision Support Systems*, 15(1):75–82, 1995.

[104] C. Shi. Approximation algorithms for stochastic optimization problems in operations management. *Wiley Encyclopedia of Operations Research and Management Science*, 2014.

[105] C. Shi, H. Zhang, X. Chao, and R. Levi. Approximation algorithms for capacitated stochastic inventory systems with setup costs. *Naval Research Logistics (NRL)*, 61(4):304–319, 2014.

[106] E. Silver. Inventory control under a probabilistic time-varying, demand pattern. *Aiie Transactions*, 10(4):371–379, 1978.

[107] E.A. Silver and H.C. Meal. A heuristic for selecting lot size quantities for the case of a deterministic time-varying demand rate and discrete opportunities for replenishment. *Production and inventory management*, 14(2):64–74, 1973.

[108] D. Simchi-Levi, X. Chen, and J. Bramel. *The logic of logistics: theory, algorithms, and applications for logistics management*. Springer Science & Business Media, 2013.

[109] D-P. Song. Optimal threshold control of empty vehicle redistribution in two depot service systems. *Automatic Control, IEEE Transactions on*, 50(1):87–90, 2005.

[110] D-P. Song. Characterizing optimal empty container reposition policy in periodic-review shuttle service systems. *Journal of the Operational Research Society*, 58(1):122–133, 2007.

[111] D-P. Song and J-X. Dong. Empty container repositioning. In *Handbook of Ocean Container Transport Logistics*, pages 163–208. Springer, 2015.

[112] D-P. Song and C.F. Earl. Optimal empty vehicle repositioning and fleet-sizing for two-depot service systems. *European Journal of Operational Research*, 185(2):760–777, 2008.

[113] D-P. Song and Q. Zhang. A fluid flow model for empty container repositioning policy with a single port and stochastic demand. *SIAM Journal on Control and Optimization*, 48(5):3623–3642, 2010.

[114] D-P. Song and Q. Zhang. 14 optimal inventory control for empty containers in a port with random demands and repositioning delays1. *International Handbook of Maritime Economics*, page 301, 2011.

[115] C.R. Sox. Dynamic lot sizing with random demand and non-stationary costs. *Operations Research Letters*, 20(4):155–164, 1997.

[116] S.A. Tarim, B. Hnich, R. Rossi, and S. Prestwich. Cost-based filtering techniques for stochastic inventory control under service level constraints. *Constraints*, 14(2):137–176, 2009.

[117] S.A. Tarim and B.G. Kingsman. The stochastic dynamic production/inventory lot-sizing problem with service-level constraints. *International Journal of Production Economics*, 88(1):105–119, 2004.

[118] S.A. Tarim and B.G. Kingsman. Modelling and computing (r n, s n) policies for inventory systems with non-stationary stochastic demand. *European Journal of Operational Research*, 174(1):581–599, 2006.

[119] R.H. Teunter, Z.P. Bayindir, and W.V. Den Heuvel. Dynamic lot sizing with product returns and remanufacturing. *International Journal of Production Research*, 44(20):4377–4400, 2006.

[120] H. Tunc, O. A Kilic, S.A. Tarim, and B. Eksioglu. A reformulation for the stochastic lot sizing problem with service-level constraints. *Operations Research Letters*, 42(2):161–165, 2014.

[121] W. Van den Heuvel. On the complexity of the economic lot-sizing problem with remanufacturing options. Technical report, Econometric Institute Research Papers, 2004.

[122] W. van den Heuvel. *The economic lot-sizing problem: New results and extensions*. Erasmus Research Institute of Management (ERIM), 2006.

[123] W. Van den Heuvel and A.P.M. Wagelmans. Worst-case analysis for a general class of online lot-sizing heuristics. *Operations research*, 58(1):59–67, 2010.

[124] S. Van Hoesel, H.E. Romeijn, D.R. Morales, and A.P.M. Wagelmans. Integrated lot sizing in serial supply chains with production capacities. *Management Science*, 51(11):1706–1719, 2005.

[125] V. Vargas. An optimal solution for the stochastic version of the wagner–whitin dynamic lot-size model. *European Journal of Operational Research*, 198(2):447–451, 2009.

[126] A.F. Veinott Jr. Minimum concave-cost solution of leontief substitution models of multi-facility inventory systems. *Operations Research*, 17(2):262–291, 1969.

[127] A. Wagelmans, S. Van Hoesel, and A. Kolen. Economic lot sizing: an o (n log n) algorithm that runs in linear time in the wagner-whitin case. *Operations Research*, 40(1-supplement-1):S145–S156, 1992.

[128] H.M. Wagner and T.M. Whitin. Dynamic version of the economic lot size model. *Management science*, 5(1):89–96, 1958.

[129] M.R. Wagner. Online lot-sizing problems with ordering, holding and shortage costs. *Operations Research Letters*, 39(2):144–149, 2011.

[130] N-M. Wang, Z-W. He, J-C. Sun, H-Y. Xie, and W. Shi. A single-item uncapacitated lot-sizing problem with remanufacturing and outsourcing. *Procedia Engineering*, 15:5170–5178, 2011.

[131] U. Wemmerlöv. The behavior of lot-sizing procedures in the presence of forecast errors. *Journal of Operations Management*, 8(1):37–47, 1989.

[132] W.W. White. Dynamic transshipment networks: An algorithm and its application to the distribution of empty containers. *Networks*, 2(3):211–236, 1972.

[133] Linwei Xin and David A Goldberg. Optimality gap of constant-order policies decays exponentially in the lead time for lost sales models. *arXiv preprint arXiv:1409.1499*, 2014.

[134] J. Yang, B. Golany, and G. Yu. A concave-cost production planning problem with remanufacturing options. *Naval Research Logistics (NRL)*, 52(5):443–458, 2005.

[135] L. Yi. *Maritime empty container repositioning with inventory-based control*. PhD thesis, 2012.

[136] L. Yin. *Operational model for empty container repositioning*. PhD thesis, 2012.

[137] W.I. Zangwill. Minimum concave cost flows in certain networks. *Management Science*, 14(7):429–450, 1968.

[138] W.I. Zangwill. A backlogging model and a multi-echelon model of a dynamic economic lot size production system-a network approach. *Management Science*, 15(9):506–527, 1969.

[139] B. Zhang, C.T. Ng, and T.C.E. Cheng. Multi-period empty container repositioning with stochastic demand and lost sales. *Journal of the Operational Research Society*, 65(2):302–319, 2013.

[140] M-J. Zhang, S. Küçükyavuz, and H. Yaman. A polyhedral study of multiechelon lot sizing with intermediate demands. *Operations research*, 60(4):918–935, 2012.

[141] P. Zipkin. On the structure of lost-sales inventory models. *Operations Research*, 56(4):937–944, 2008.

[142] Paul Zipkin. Old and new methods for lost-sales inventory systems. *Operations Research*, 56(5):1256–1263, 2008.

[143] Paul Herbert Zipkin and Paul H Zipkin. *Foundations of inventory management*. 2000.

Appendix A

Scientific Career

Jul 2013 - Sep 2016	PhD Candidate in Mathematics at the Technical University of Kaiserslautern, Germany (funded by a Scholarship of the Fraunhofer ITWM).
Jan 2013 - Jun 2013	Project Studies in Advanced Technology (ProSat) at the Technical university of Kaiserslautern, Germany (funded by a scholarship of the Fraunhofer Institute for Industrial Mathematics (ITWM))
Jul 2012 - Nov 2012	Engineer in Optimization at *Eurodecision*, Versailles, France.
Oct 2010 - Jun 2012	Diploma Study in Computer Science at the University of Karlsruhe *(KIT)*, Germany. Title of the Diploma thesis: *Point Labeling with Leaders for Convex Boundaries.*
Sep 2008 - Sep 2010	Diploma Study in Computer Science and Applied Mathematics at the engeneering graduate school *Grenoble INP - Ensimag* of Grenoble, France.
Sep 2006 - Jun 2008	Higher School Preparatory Classes in mathematics *(MP*)* at the Highschool *La Martiniere Monplaisir*, Lyon, France.

Appendix B

Akademischer Werdegang

Jul. 2013 - Sep 2016	Doktorand in Mathematik an der Technischen Universität Kaiserslautern, Deutschland (gefördert durch ein Stipendium des Fraunhofer ITWM).
Jan 2013 - Jun 2013	Project Studies in Advanced Technology (ProSat) an der Technischen Universität Kaiserslautern, Deutschland (gefördert durch ein Stipendium des Fraunhofer ITWM).
Jul 2012 - Nov 2012	Ingenieur in Optimierung bei *Eurodecision*, Versailles, Frankreich.
Okt 2010 - Jun 2012	Diplom Informatik am Karlsruher Institut für Technology *(KIT)*, Deutschland. Titel der Diplomarbeit: *Point Labeling with Leaders for Convex Boundaries*.
Sep 2008 - Sep 2010	Diplom in Informatik und angewandter Mathematik an *Grenoble INP - Ensimag* in Grenoble, Frankreich.
Sep 2006 - Jun 2008	Vordiplom "classes préparatoires" in Mathematik *(MP*)* an *La Martiniere Monplaisir*, Lyon, Frankreich.

Appendix C

Publications

The following publications contain parts of this thesis or precursory work:

[53] N Jami, M Schröder. Tactical and Operational Models for the Management of a Warehouse. *Dynamics in Logistics*:655–665, 2016.

[101] M Schröder, N Jami, U Beissert, M Motta. Konzeptionierung eines integrierten modellbasierten Ansatzes zur Prognose von transportlogistischen und intralogistischen Ereignissen in Logistiknetzwerken. *Tagungsband der 16. ASIM Fachtagung Simulation in Produktion und Logistik*, 23.-25.09.2015, Dortmund, 2015.

[54] N Jami, M Schröder, KH Küfer. A Model and Polynomial Algorithm for Purchasing and Repositioning Containers. *7th IFAC Conference on Management and Control of Production and Logistics*. Proceedings of the MCPL 2016.

[55] N Jami, M Schröder, KH Küfer. Online and Offline Container Purchasing and Repositioning Problem. *7th International Conference on Computational Logistics*, 2016.

The objective of this thesis is to develop models and algorithms to plan the purchasing of reusable containers in a closed-loop supply chain where the demand is increasing. We restrict our study to a periodic review process between a single manufacturer and a single supplier. Each item is transported either in a reusable container or in a single-use disposable. Furthermore, a setup cost is paid every time new containers are purchased. Consequently, our model is similar to a lot-sizing problem with return of every item after a fixed duration. We study both cases of a deterministic demand as well as a stochastic demand. In the deterministic setting, we use dynamic programming and minimum linear-cost flows to generate polynomial time algorithms. When the demand is stochastic, we use the Markov decision process framework to develop pseudo-polynomial time heuristics for four different strategies. We show the L-natural-convexity of the cost functions for three strategies to speed up the computations. The thesis concludes with an application on a real-life supply chain.

ISBN 978-3-8396-1210-1

FRAUNHOFER VERLAG