Stochastic Stock Control

Comparing Stochastic Programming and Heuristic Approaches for Inventory Control Problems

Coppelmans, A.T.J.

Award date:
2019

Stochastic Stock Control

Comparing Stochastic Programming and Heuristic Approaches for Inventory Control Problems

Eindhoven University of Technology

Final Thesis

Sioux LIME

Author:
Arno Coppelmans

Supervisors:

Keith Myerscough	(Sioux LIME)
Lense Swaenen	(Sioux LIME)
Corné Suijkerbuijk	(Sioux LIME)
Sem Borst	(TU/e)

January 13, 2020

Abstract

Most modern-day issues where optimization is required involve uncertainty, in particular in the form of events which are outside of our control. Incorporating stochastic aspects in optimization problems allows one to find solutions which are more robust to these uncertainties. In particular, when focusing on inventory control problems, several different techniques exist, ranging from fully heuristic to computationally expensive decision methods. In order to compare these methods, a simulation framework is built, in which several different inventory problems can be formulated, such that techniques can be tested on multiple instances.

The two methods which were investigated in detail are stochastic programming and a base stock policy. Both methods can be used for multiple different optimization problems, but the base stock policy is particularly tailored to inventory control problems. Because of this, it is able to provide solutions which are close to optimal with only limited computation time required. Within the framework, complexities were added to test the accuracy of such a policy method, which turns out to remain quite efficient for more difficult instances.

Stochastic programming requires much longer computation time, but it provides more efficient and accurate results than the simpler heuristic algorithms. In particular, it is able to make stochastic decisions, where similar model states result in different decisions made. This results in more robustness to uncertainty, allowing for better results overall. If a shorter amount of computational time is allowed, stochastic programming can be used to optimize important one-time decisions, while using a heuristic approach for the remaining decisions.

When considering the modular expansion of the decision methods, the heuristic algorithm gives more complications than the stochastic program. When iteratively adding complexities and difficulties to the policy method, it can easily result in bloated heuristics. Stochastic programming, however, allows for easier modular expansion of simulations and models, as additional complexities or assumptions can be modified naturally. Initial investigation of this topic shows promising results for the stochastic program if computational time allows. Further research could greatly decrease this time required, while maintaining similar efficiency and accuracy for the results. By researching specialized techniques and model structures for stochastic programming, it can be used as a computationally efficient optimization method which is able to effectively incorporate uncertainty.

Contents

1 Introduction

Optimization under uncertainty is a key aspect in many modern-day issues. Faster computations and more data allow us to investigate matters in more detail, considering all aspects much more extensively. Instead of merely optimizing under a certain expectation of uncertainty, the focus can be put on incorporating uncertainty throughout the design process, such that new, more robust solutions can be found.

One of the fields in which optimization under uncertainty is required is in the field of inventory control and supply chains. Stock needs to be transported between locations in order to fulfil demand, at several different scales: a stock clerk restocking its items on display, or a long distance supplier of chemical goods. For each of these situations, decisions need to be made under uncertainty: the stock clerk might not know what the customers demand, and transport trucks might be blocked by traffic jams. But it could even be prior to the actual restocking, where decisions must be made on where warehouses are constructed, or how products are transported. Sometimes, an educated guess can be made considering how these uncertainties will unfold, in particular by using past data on similar matters. Still, one can never be absolutely sure of how things will be, such that decisions should be made which are robust for all sorts of uncertainty.

More often than not, uncertainty reveals itself in several stages, where decisions can be adjusted at each stage. In fact, decisions are often based on prior decisions and revealed uncertainty, as a better understanding often results in better decisions. For example, if there are only a limited number of stock clerks in a store, a choice must be made which items to restock at a particular day. Knowing the rate at which items are bought from the store provides a better understanding of which products to restock first. Still, waiting until shelves are completely empty before starting to restock them is often a bad idea. This can easily be prevented by restocking shelves in time. Next to that, the stock clerks can be utilized more efficiently, resulting in lower costs overall.

Research Question

The focus of thesis is on investigating and modelling these optimization problems, where dealing with uncertainty is key. Several different techniques are researched, focusing on solutions which are either computationally fast or very accurate. In particular, these techniques are compared to simpler heuristics, which have only limited incorporation of uncertainty. In short, the research question which we focus on is

What is the effect of incorporating the stochastic aspects in inventory control problems, and how do different solution techniques within a particular framework compare to one another?

1.1 Sioux Lime

The supervision of this project has primarily been performed by Sioux Lime. Lime is a consultancy agency in Eindhoven, focused on providing Mathware solutions for its customers. By combining state-of-the-art algorithms with different fields of expertise (such as Mathematics or Physics), solutions can be found in terms of either software or hardware. The company originates from the department of Mathematics & Computer Science from the Eindhoven University of Technology. Later, it separated itself, and became part of the Sioux group.

1.2 Motivation

This thesis was initiated as a research project for the optimization expertise group of Lime. Instead of focusing directly on a particular project of a customer, the goal was set to further research the field of optimization under uncertainty. This way, other projects could benefit from any knowledge and models resulting from this research project. Initially, the main goal was to look into stochastic programming, and determine how other optimization could benefit from this. As the field of stochastic programming is very broad, we have only scratched the surface of the various techniques. In addition to the techniques in the field of stochastic programming, we researched robust optimization and several (advanced) heuristics, primarily focused on inventory control problems.

Using these techniques, the next step was to create a solution method for all kinds of different optimization issues. An interesting topic to investigate would be the garbage collection in a municipality, which would require quite complex optimization to find optimal solutions. Another topic of interest was to determine a schedule or algorithm for picking robots, which would be used to pick items from a conveyor belt and place them in separate boxes. With these separate issues in mind, our goal was to find similarities between these separate problems: we found that restocking inventory in a warehouse had great similarities to an inventory routing problem. These in turn could be compared to a much simpler newsvendor problem; as such, by using minor simplifications or assumptions, several optimization issues could be modelled and solved with the same framework. By starting from the basic newsvendor problem, we were able to add complexities and reformulations to use the same idea for the optimization of inventory in a warehouse. With this in mind, the focus was to construct such a general framework, where we would be able to easily modify several model aspects.

With a general idea of a framework in place, the next step in the process was to construct a model, able to create a simulation to compare the various techniques. The model was constructed with the idea of optimizing inventory in a warehouse in mind, but in such a way that it could be used for different optimization problems as well. By using simulation techniques from the fields of stochastics (from queueing theory in particular), we were able to construct such a model. It formulates a linear program to indicate which constraints should be taken into account, and uses predictions of uncertainty to determine better decisions. Using this model, we were able to create a simulated environment, in which the different techniques could be compared to one another, where each environment could represent a different optimization issue.

The final leg of the research project consists of a more in-depth investigation of simulation results, in particular to compare the different methods used to find (optimal) decisions. For this end, several examples of inventory control in a warehouse are described, where different aspects of each method are highlighted. Next to this, the same framework is used for a larger problem in the field of inventory routing. The solution techniques are tested in such a way that we have an idea on how they would perform in real-life optimization cases.

1.3 Problem Formulation

The main problem that is investigated in this report is the inventory control of a (fictive) warehouse. Several different products are kept in stock, which are used to fulfil orders from customers. To increase efficiency, the warehouse is split into two main parts: the direct- and bulk-pick area. The direct-pick area is a small and compact area, where items are stored in such a way that they can easily be retrieved by the order pickers. The bulk-pick area, however, denotes the spacious zone where all items are stored on pallets reaching from the bottom up to the ceiling. By assumption, the inventory level (i.e. the number of items in stock) of the bulk-pick area is 'unlimited', in the sense that there is never a shortage of products.[1] The goal of this split in zones is so that products are easily picked in the direct-pick area, which in turn is refilled from the bulk-pick area.

Throughout the day, customers place orders for specific products. The cumulation of all orders on a day is denoted as the demand. Each day, this demand needs to be fulfilled entirely; preferably, all items are taken

[1]While this may seem to be a strong assumption, the bulk-pick area is restocked with products over time through external deliveries. As such, there are always enough products to refill the direct-pick area.

from the direct-pick area, as this is much more time-efficient. However, if an item is no longer available in the direct-pick area, it must be retrieved from the bulk-pick area. Doing this takes a significantly longer time, as heavy machinery is needed to access items in the bulk-pick area.

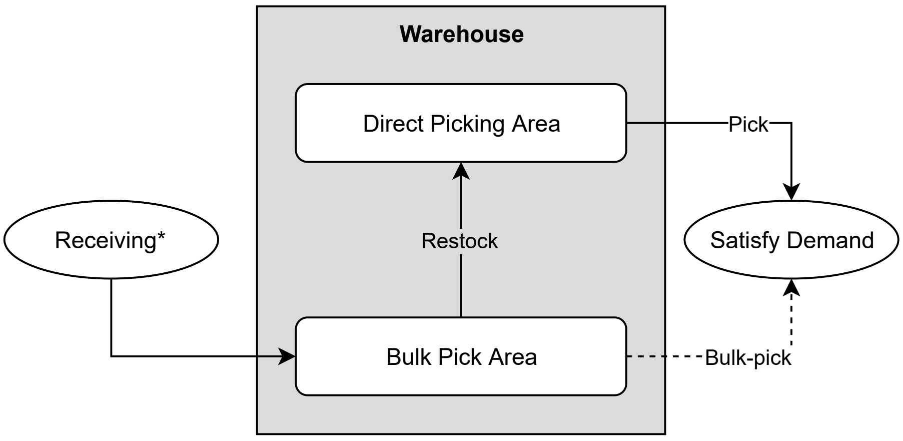


Figure 1.1: Schematic view of the warehouse described. Note that the receiving of products is taken for granted, and is not further investigated in this report.

As the direct-pick area is more time-efficient (considering the picking of orders) than the bulk-pick area, an efficient strategy is to restock products from the bulk- to the direct-pick area. Restocking is performed either at specific discrete times (i.e. at the start of each day), or continuously throughout the day. For this report, each day starts with a restocking possibility, with no restock moments throughout the day. When items are restocked, they are taken from the bulk-pick area (which is assumed to have enough stock), and placed in the direct-pick area. Unfortunately, restocking is a time-consuming activity as well: items need to be retrieved from the bulk-pick area, and subsequently placed in the direct-pick area using a First-In-First-Out policy (FIFO). Essentially, this means that restocking takes more time if there are more items in the direct-pick area: items need to be taken out or at least moved, such that the newer products can be placed behind them. If a restock is performed with too many items in stock, then the usage of a FIFO policy gives a significant overhead; restock too late, and we might need to perform a bulk-pick action too often. As such, we are interested in the optimal moment to restock: the restock action needs to be worth the required time, while maintaining a low probability of needing to perform a bulk-pick action.

Figure 1.2: View of a bulk-pick area.

1.4 Report Outline

In order to solve a multi-stage decision problem with uncertainty, different techniques have been researched in the field of stochastic programming and robust optimization (Chapter 2). Using these techniques, the warehouse in question is modelled (Chapter 3). In order to model uncertainty (in the form of demand) each day, several different methods were investigated and worked out (Chapter 4). Lastly, a simulation is used (Chapter 5) to investigate the effects of different techniques, and results for several examples are provided (Chapter 6). Lastly, we conclude and discuss our findings, and provide pointers for possible future research (Chapter 7).

2 | Literature Research

The techniques described in this chapter are all aimed at incorporating uncertainty or randomness into the optimization of inventory problems. This has the advantage that solutions are more robust to the uncertainty in our model; however, finding these solutions is often a computationally heavy task. Most of the techniques described require solving some Mixed Integer Linear Program (MILP). These are solved using either a freely or commercially distributed linear program solver. Next to this, the techniques described are easily extendable to different instances; for this report, the focus is primarily on controlling the inventory levels of a warehouse. In the remainder of this chapter, the basics of several techniques are described, including algorithms and heuristics aimed at decreasing computational times while maintaining sufficient accuracy.

2.1 Newsvendor Model

The inventory model in its most basic form is the newsvendor model, where the mathematical model dates back to an older publication from Edgeworth.[Edgeworth, 1888] Here, he determines optimal cash reserves in order to satisfy uncertain withdrawals. The model is used to determine the number of products to purchase, given the price to purchase and sell, where the demand is uncertain. As an analogy, imagine a newsboy who has to decide each morning how many papers to purchase in stock, not knowing the exact demand on a day, where any unsold copies are worthless at the end of the day. The newsvendor model is a basic single-stage inventory model, as only a single decision needs to be made. For multiple successive days, each decision is independent of any past decisions or demands (assuming the demand is independent in time).

Let D be the random variable denoting the demand, p the purchase price per product, s the selling price, and q the number of products purchased. It is assumed that $p < s$, otherwise the optimal decision would be purchasing no papers at all. This profit function which we wish to maximize is given by

$$\begin{aligned} \max_{q} \quad & \mathbb{E}[s \cdot \min(q, D)] - pq \\ \text{s.t.} \quad & q \geq 0 \end{aligned} \tag{2.1}$$

Observe that, if the number of items bought q is less than the demand D, only q items can be sold. On the other hand, the number of items sold cannot exceed the demand. In any case, the price p has to be paid for every item purchased. As explained in the remainder of this section, the optimal order amount q^* can easily be determined:

$$q^* = F_D^{-1}\left(\frac{s-p}{s}\right) \tag{2.2}$$

Intuitively, the optimal order amount increases if the ratio between the profit per product $s - p$ and purchase price p increases. This optimal order amount can be found by conditioning on the demand being higher or lower than the order amount:

$$\begin{aligned}
\mathbb{E}[s \cdot \min(q, D)] - pq &= \mathbb{P}(D \leq q) \cdot \mathbb{E}[s \cdot D \mid D \leq q] + \mathbb{P}(D > q) \cdot \mathbb{E}[sq \mid D > q] - pq \\
&= F_D(q) \cdot s \cdot \frac{\int_{x \leq q} x f_D(x) \cdot dx}{\int_{x \leq q} f_D(x) \cdot dx} + sq \cdot (1 - F_D(q)) - pq \\
&= s \cdot \int_{x \leq q} x f_D(x) \cdot dx + sq \cdot (1 - F_D(q)) - pq \qquad (2.3)
\end{aligned}$$

Taking the derivative with respect to q gives

$$\begin{aligned}
\frac{\delta}{\delta q} \mathbb{E}[\text{profit}] &= sq \cdot f_D(q) + s \cdot (1 - F_D(q)) + sq \cdot (-f_D(q)) - p \\
&= s(1 - F_D(q)) - p \qquad (2.4)
\end{aligned}$$

Setting the derivative to 0 and solving for q gives

$$\implies q^* = F_D^{-1}\left(\frac{s-p}{s}\right) \qquad (2.5)$$

In the remainder of this section, methods are described to extend this basic newsvendor problem into a model which can be used for our inventory control problem.

2.1.1 Two-Stage Newsvendor

The first way to extend the simple one-stage newsvendor problem is to add a second decision moment, dependent on the first. The difference between these decision moments is that the first is made before the realization of the uncertainty, but the second is made afterwards. In general, the second decision not only depends on the uncertainty, but also on the prior decision(s). To extend the model from above, let c be the salvage profit: any papers not sold by the end of the day can be salvaged for c per item. Logically, assume that $c < p$, otherwise the optimal order amount would be infinitely large. Let $y(q, D)$ be the second stage decision indicating the number of papers to salvage. The mathematical formulation of this adapted model is given by

$$\begin{aligned}
\max_{q,\, y(q,D)} \quad & -pq + \mathbb{E}[s \cdot \min(q, D) + c \cdot y(q, D)] \\
\text{s.t.} \quad & q \geq 0 \\
& 0 \leq y(q, D) \leq \max(0, q - D)
\end{aligned} \qquad (2.6)$$

Observe that if the demand exceeds the number of items in stock, then no items can be salvaged at the end of the day. While this extension of the model adds a second stage decision to the model, the solution is still the same: the optimal order quantity q remains as Equation (2.2), and $y(q, D)$ must be chosen maximally, such that all remaining papers are salvaged. One possible way of actually increasing the difficulty of solving this problem is to have uncertain prices. Finding the optimal order quantity q becomes significantly more difficult if the prices p, s and c are no longer fixed. In particular, if $\mathbb{P}(s < c) > 0$, it is possible that salvaging items is worth more than selling them. This changes the obvious choice of selling all items in stock up to the demand, as items might be worth more if they are salvaged instead of sold. In fact, one can even alter the moment when this uncertain salvage price c is revealed, in particular before or after determining the number of items to salvage.

General Two-Stage Decision Problem

The objective value of the two-stage newsvendor problem follows the basic formulation of a two-stage decision problem: let x_t be the decision vector and f_t the objective value at stage t (for now, $t \in \{1, 2\}$), and let

$\xi_{[t]} := (\xi_1, \xi_2, ..., \xi_t)$ represent the history of uncertainty up to time t. Then the objective value is formulated by

$$\max_{x_1 \in X_1} f_1(x_1) + \max_{x_2 \in X_2(x_1, \xi_1)} \mathbb{E}[f_2(x_2, \xi_1)] \tag{2.7}$$

Note that each decision variable x_t has to be an element of the set $X_t(x_{t-1}, \xi_{[t-1]})$, which indicates that decisions and realizations of the uncertainty affect the current set of possible decisions. In the newsvendor example, the number of salvaged papers $y(q, D)$ is a second-stage decision, influenced directly by the number of papers bought and sold.

2.1.2 Extension to Multiple Stages

Another model extension to the newsvendor problem is to view multiple days in succession, where decisions are in fact influenced by previous ones. For example, one could replace the papers by products which can be kept in stock for longer periods of time. This way, any products at day's end can be stored until the next day. For these products, a holding cost h has to be paid. Following the formulation of (2.6), assume that the salvage profit $c \equiv 0$ (as products will be sold eventually), and define the order quantity at day t by

$$x_t := f\left((x_1, x_2, ..., x_{t-1}), (D_1, D_2, ..., D_{t-1})\right) \tag{2.8}$$

The decision at day t is some function $f(\cdot)$ depending on all decisions and (realizations of) demands up to day $t-1$. For clarity, define y_t as the number of products in stock at the start of day t, and assume $y_1 = 0$. For a given time-period T, the objective is to maximize expected profit, given by

$$\begin{aligned} \max_{\{x_t\}_{t=1}^T} \quad & \sum_{t=1}^{T} -p \cdot x_t + \mathbb{E}\left[s \cdot \min(y_t + x_t, D) - h \cdot y_{t+1}\right] \\ \text{s.t.} \quad & x_t \geq 0 \qquad\qquad\qquad\qquad \forall t \in \{1, ..., T\} \\ & y_{t+1} = \max\left(y_t + x_t - \widetilde{D}_t, 0\right) \quad \forall t \in \{1, ..., T\} \end{aligned} \tag{2.9}$$

Here, $\widetilde{D}_t$ denotes the realized demand at day t. As the inventory level at each day is dependent on the (uncertain) demand, the goal is to minimize the average holding costs. Additionally, note that there are no holding costs if there are no items in stock.

If the demand realizations are independent and identically distributed (i.i.d.) throughout the horizon, it can be noted that the inventory should be restocked to some level $\Omega \geq 0$ on a daily basis. As the objective value is not influenced by how often a restock is performed, there is no reason not to restock every day. Next to that, as the (distribution of the) demand is identical every day, the optimal inventory level is the same every day. In other words, the optimal restock amount is given by

$$x_t^* := \Omega - y_t \tag{2.10}$$

This formula can be plugged into the equations of (2.9); however, depending on the parameter settings p, s, h and the exact distribution of D, different values of Ω are found. We will not go into further detail on this for the newsvendor model, but more on this phenomenon can be found in Section 2.4.

General Multi-Stage Decision Problem

The objective value formulation as seen in (2.7) can be extended to multiple stages, using similar reasoning. For T stages, the objective value can be formulated as

$$\min_{x_1 \in X_1} f_1(x_1) + \mathbb{E}\left[\min_{x_2 \in X_2(x_1, \xi_1)} f_2(x_2, \xi_1) + \mathbb{E}\left[... + \mathbb{E}\left[\min_{x_T \in X_T(x_{T-1}, \xi_{[T-1]})} f_T(x_T, \xi_{[T-1]})\right]\right]\right] \tag{2.11}$$

The nested expectations do not appear to occur in the multi-stage newsvendor problem (2.9), but these are circumvented by explicitly defining the inventory level y_t. Unfortunately, because of these nested expectations, analytically solving a general multi-stage decision problem is often not possible. The remainder of this chapter will describe different techniques in order to numerically solve such problems.

2.2 Stochastic Programming

The first and foremost technique investigated in this report is stochastic programming. [Shapiro and Philpott, 2007] The main goal of stochastic programming is to incorporate the uncertainty of (random) variables into the optimization steps, such that the (optimal) solution is robust considering the uncertainty. Essentially, the randomness is discretized into separate scenarios. For each scenario, additional variables and constraints are formulated, which are in turn solved using a linear program. Single-stage decision problems can often be solved using an analytical approach (such as the newsvendor problem), yet only limited theoretical solutions are known for multi-stage stochastic decision problems. Because of this, stochastic programming is a widely used technique for solving complex multi-stage problems. [Boland et al., 2008] [Kim et al., 2015]

Stochastic programming is based on creating a scenario tree, which is a projection of (a subset of) the uncertainty. An example scenario tree for a two-stage decision problem is displayed in Figure 2.1. Here, levels 1 and 2 of the tree consist of two new scenarios, each with a specific probability of occurring. Starting at the root node, we are able to compute the probability to end up in leaf node $(1, 2)$ by $p \cdot (1 - p_1)$, and likewise for the other leaves.

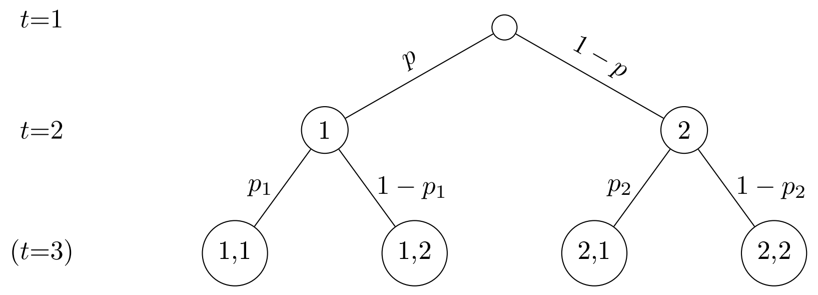


Figure 2.1: An example scenario tree for $T = 2$ stages. Each node represents a series of scenarios up to that point, and each edge the probability of that particular scenario. The new scenarios originating from the leaf nodes at $t = 3$ are not taken into account.

In order to model such a scenario tree, a requirement is being able to sample realizations from the uncertain variables, i.e. using the distribution of a random variable. If more detailed information of the uncertainty is known (e.g. the support or even the probability density function), other scenario generating techniques can be used. Several examples of scenario-generation are as follows:

Unbiased sampling

Generate n samples of the uncertainty at each stage, where each sample corresponds to a different scenario. Note that this sampling technique does not prevent identical scenarios originating from a single node.

Percentile discretization

Generate n equidistant numbers on the interval $[0, 100]$, e.g. $\frac{k}{n+1}$ for $k = 1, 2, ..., n$. Then, for each of these numbers v, compute the v'th percentile of the random variables, and use these for the new scenarios.

Priority sampling

Generate samples according to a predefined priority function. For example, samples are generated close to the extremes (tails) of the distribution, or with high probability above the expected value.

Note that for higher dimensional random variables, the sampling techniques might be altered to sample separately for each dimension. For example, the percentile discretization is performed separately for each dimension. As such, if we wish to use a percentile discretization of n steps, where the uncertainty is m-dimensional, there are m^n scenarios for a single stage.

Having created a scenario tree, which contains a discretization of the uncertainty, we are left with a deterministic multi-stage optimization problem. By creating additional decision variables and constraints for each of these scenarios, a much larger linear program is constructed.

Following the general multi-stage formulation (2.9), as described in Section 2.1.2, the objective function with respect to the scenario tree of Figure 2.1 is reformulated as

$$\min_{x_1 \in X_1} f_1(x_1) + p \cdot \left(\min_{x_2 \in X_2(x_1, \xi_1 = 1)} f_2(x_2, \xi_1 = 1) \right) + (1 - p) \cdot \left(\min_{x_2 \in X_2(x_1, \xi_1 = 2)} f_2(x_2, \xi_1 = 2) \right) \tag{2.12}$$

While not directly visible in the equations above, both functions $f_2(x_2, \xi_1)$ can use the predictions of the upcoming scenarios at $t = 3$. However, at each node, only a single decision x_t can be made, regardless of how many scenarios originate from that node. More on this in the next section.

Unless the scenario tree incorporates *all* possible scenarios, it cannot be shown that the optimal decisions found by solving (2.12) are the true optimal decisions. To circumvent this, more scenarios can be added. This decreases the optimality gap, yet significantly increases the computation time required to solve the model.

Nonanticipativity Constraints

When formulating the decision variables of a scenario tree, one must be cautious: at each node in the scenario tree, only a single decision can be made, regardless of what the upcoming scenarios are. Decision variables x need to satisfy the *nonanticipativity constraints*: all decisions with identical past scenarios must be equal, regardless of the upcoming scenarios.

A commonly used manner of formulating all decision variables is by using a split-variable notation. [Defourny et al., 2011] Instead of defining a single decision variable per node at each stage, each stage has the same number of variables. Following our previous example (Figure 2.1), there are four leaf nodes, defined as the final scenarios A, B, C, D. Then, at each stage t, 4 decision variables $x_{t,A}, x_{t,B}, x_{t,C}$ and $x_{t,D}$ are defined, as shown in Figure 2.2.

In Figure 2.2, notice that the root node has been split into four separate variables. By nonanticipativity, these four variables must all be the same value: the decision cannot depend on the future, as it is not possible to know which of the four end-scenarios will occur. The same logic is used for the second-stage variables, such that the following constraints need to be added:

$$\begin{gathered} x_{1,A} = x_{1,B} = x_{1,C} = x_{1,D} \\ x_{2,A} = x_{2,B} \qquad x_{2,C} = x_{2,D} \end{gathered} \tag{2.13}$$

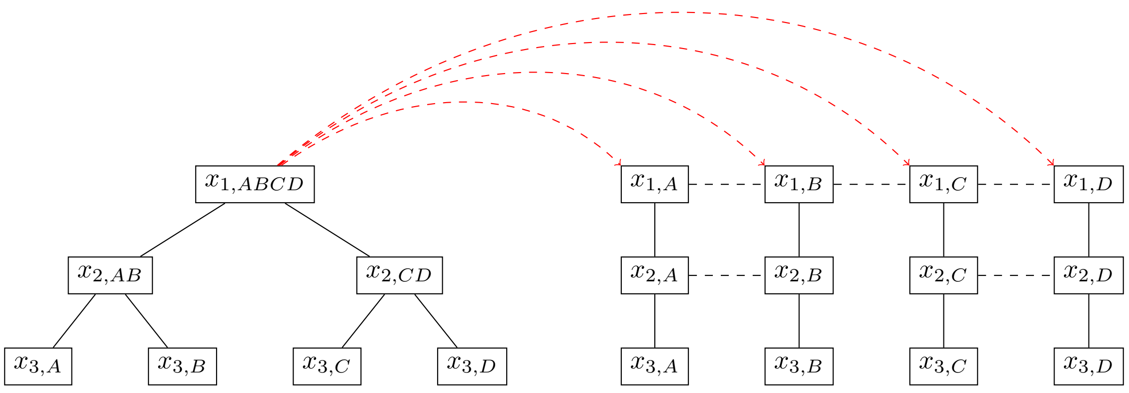


Figure 2.2: Schematic view of the conversion to split-variable notation with nonanticipativity constraints, following the example of Figure 2.1. On the right, horizontal lines indicate the nonanticipativity constraints between decision variables.

Because of the large number of constraints which are added, the split-variable notation is not an interesting formulation method for us. However, several algorithms (such as the progressive hedging algorithm, see Section 2.2.2) require a split-variable notation, and it allows for easier (array) notation of decision variables.

2.2.1 Receding Horizon

When optimizing decision variables considering a large number of scenarios, stochastic programming requires solving a large (mixed-integer) linear program, as the number of decision variables grows exponentially with the number of scenarios. For example, if we wish to investigate a large number of successive stages (say $T = 50$), the linear program becomes too large to solve directly.[1] As such, a commonly used technique of a receding horizon is used. Here, instead of constructing a large, single scenario tree, we construct several smaller scenario trees over time, as illustrated in Figure 2.3. [Beltran-Royol et al., 2010]

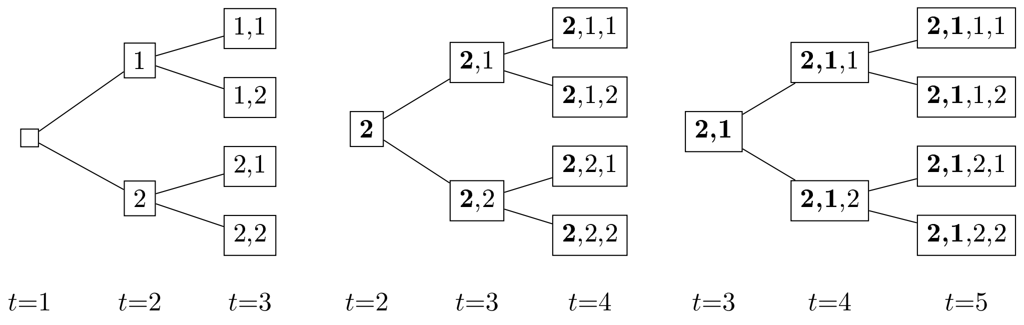


Figure 2.3: An example receding time horizon of 3 successive realizations, with a scenario tree of $T = 2$ stages at every step. Each node displays the (predicted) scenarios up to that point in time. The demand realizations are indicated in bold, where the first realization is scenario 2, and the second is scenario 1. The probabilities of each scenario are omitted for clarity.

At each time $t = 1, 2, ...$ a (small) scenario tree is constructed, based on the information currently available. In

[1] Following the example of Figure 2.1 with 2 scenarios per stage, a 50-stage stochastic program would require solving a linear program with about 10^{13} decision variables.

the example figure, it is known at time $t = 2$ that the first scenario $\xi_1 = 2$, and at time 3, it is known that $\xi_{[2]} = (2, 1)$. Each scenario tree is used to find the optimal decision at the current time (i.e. using stochastic programming). After resolving this decision, the 'true' demand becomes clear, the time t is increased, and the process is repeated.

Note that, unlike Figure 2.3 suggests, it is possible that the true demand is not one of the generated scenarios in the scenario tree. In fact, if the scenario tree consists of only a few samples of the demand, chances are that the realized scenario is different than any of the generated scenarios. Because of this, any optimal decisions found in prior scenario trees often become irrelevant, and the process of finding the optimal decision needs to be repeated at each new step.

2.2.2 Progressive Hedging Algorithm

Another heuristic approach for solving a stochastic program is to use the progressive hedging algorithm. [Rockafellar and Wets, 1991] This algorithm finds (near-)optimal values for all decision variables in a complete scenario tree by using a split-variable notation, but ignoring the nonanticipativity constraints. Because of this, we are effectively able to find all optimal decision variables per end-scenario in parallel. After having found these optimal decision variables, nonanticipativity is enforced by means of a penalty factor, which is increased at every iteration. Convergence is achieved if the separate decision variables are less than ε distance apart. For continuous decision variables, it is provable that the algorithm converges in linear time. [Rockafellar and Wets, 1991] An implementation of the progressive hedging algorithm (for continuous variables) is provided in Appendix A.2. In our case, however, we are mostly interested in discrete decision variables. While there are methods to adjust the algorithm for discrete decision variables [Watson and Woodruff, 2011], these are not in the scope of this thesis.

2.3 Robust Optimization

The second field of techniques we investigated are those in the field of robust optimization. [Ben-Tal et al., 2009] [Gorissen et al., 2015] In contrast to stochastic programming, robust optimization does not assume **any** distribution or probability measure on the uncertain data. Instead, it maximizes the objective function, such that the minimum value over all realizations of the uncertainty is maximized. For this end, a bounded uncertainty set is required for the randomness, which can be seen as the support of the random variables. Several distinctions between constructing uncertainty sets are compared in Section 2.3.1.

With only minor adjustments, the framework of stochastic programming can be used in robust optimization. In order to do so, two adjustments are made: firstly, the objective value is adjusted from minimizing an expected value to minimizing the maximum value over all scenarios. Secondly, a percentile-based scenario generation is used, where merely the minimum and maximum values of the uncertainty are sampled. It is only required to take these two samples, as it is known that the objective value is maximal when the uncertainty is either maximal or minimal - in our case, the total costs are obviously maximal if the demand at each day is maximal.

2.3.1 Definition of the uncertainty set

As the uncertainty set is the main aspect of robust optimization which can be altered, we provide a few different options for an uncertainty set. [Ben-Tal et al., 2009] For ease of display, let the number of dimensions be $n = 2$, such that the uncertainty set $\Omega \subset \mathbb{R}^n$. Note that the same reasoning holds for any dimension $n \in \mathbb{N}^+$.

Let $d = (d_1, d_2) \in \Omega$ be an element of the uncertainty set, and let γ be the (chosen) centre value.[2] As the set is bounded, there exist lower and upper bounds $(l_1, u_1), (l_2, u_2)$, such that

$$l_i \leq d_i \leq u_i \qquad \forall i \in [n] \tag{2.14}$$

[2]If the expectation of the random variables is known, this would be the optimal choice for the centre value.

Using these bounds, the box set is constructed:

$$\Omega_1 = \{d \in \mathbb{R}^n : l_i \leq d_i \leq u_i \quad \forall i \in [n]\} \tag{2.15}$$

The box set indicates that all random variables can attain their maximum and minimum values, independently of one another. However, this means that it is an extremely conservative approach to robust optimization: it might not be realistic that *all* random variables attain their maximal (or minimal) value at the same time. As such, we investigate different, smaller sets with similar properties.

The second uncertainty set is the ball-box set, which is found by taking the intersection of a ball (sphere) of radius $r \in \mathbb{R}^+$ around the centre value γ and the original box set:

$$\Omega_2 = \Omega_1 \cap \{d \in \mathbb{R}^n : \|d - \gamma\|_2 \leq r\} \tag{2.16}$$

The radius r can be chosen such that the extreme points of the box are removed from the uncertainty set. This creates a less robust solution for the extreme cases of uncertainty, but can provide (on average) much better results.

The third construction is the budgeted-box set, which uses the $\|\cdot\|_1$ norm (or taxicab metric) as a means of 'budgeting' the variation of uncertainty by $\beta \in \mathbb{R}^+$ from the centre value. Again, the set is constructed by taking the intersection with the original box set:

$$\Omega_3 = \Omega_1 \cap \{d \in \mathbb{R}^n : \|d - \gamma\|_1 \leq \beta\} \tag{2.17}$$

Similar to the ball-box set, the choice of β greatly influences the size of the uncertainty set. Similarly, it aims at removing the extreme points of the uncertainty set. An example of how these sets are constructed is provided in Figure 2.4

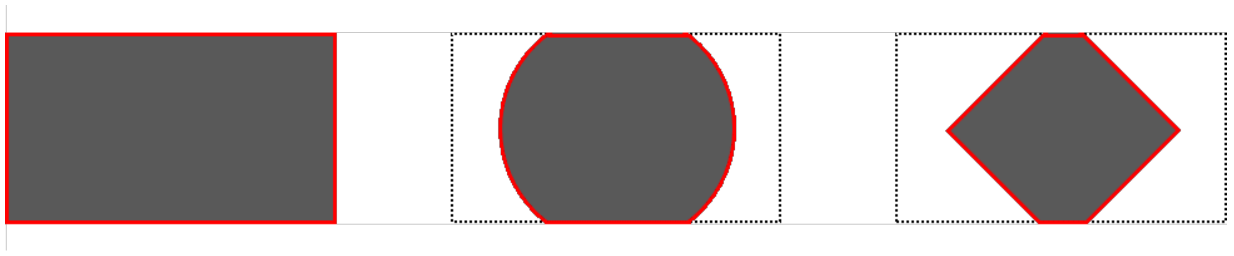


Figure 2.4: The three uncertainty sets, from left to right: box, ball-box, budgeted-box. Note that in the latter two, only the overlapping part is considered as the uncertainty set.

For our goal of optimizing an inventory restocking process, initial solutions resulting from robust optimization techniques appear to be too conservative; as the process consists of repeating events and decisions, it is much more interesting to have solutions which perform better on average, instead of performing optimal in the worst-case scenarios. This could be researched further with the improvement of different uncertainty sets, but this is outside of the scope of this thesis.

2.4 Base Stock Policy

The third technique we investigated is the use of a base stock policy. This technique is often used in inventory control problems, as it is computationally extremely light to determine which products require a restock, while providing optimal results under certain assumptions. [Sethi and Cheng, 1997] The policy which we investigate is the (s, S) policy, which can be described as follows: let x_t be the decision variable indicating the number of items to restock at stage t, and y_t the inventory level at stage t. Then

$$x_t = \begin{cases} 0 & \text{if } y_t \geq s \\ S - y_t & \text{else} \end{cases} \tag{2.18}$$

Essentially, a restock only occurs if the inventory level is below the threshold s. The advantage of using this technique is that, if the policy values (s, S) are known, then it is merely a simple comparison to decide how much to restock. The disadvantage, however, is that the base stock policy does not allow any freedom of choice throughout the process, as restocks are only performed if the inventory is below the threshold s. Next to that, finding the 'optimal' policy values might be quite difficult, and choosing sub-optimal policy values might result in either far too many restocks or inventory shortages.

2.4.1 Finding the Policy Values

Both policy values (s, S) can be found using several different methods. A different method can be used, depending on how much computation time is allowed, the size of the support of both s and S, and how close to optimal the policy values need to be. Several methods are described in the remainder of this section, each excelling at different aspects.

Heuristic Average

The fastest method which we investigate is to heuristically determine the best average restock level. To use this method, the density function of the uncertainty is required. In addition, the value of S is set to be maximal: only a single stage is considered, and choosing a very low value of S provides much better results when considering only a single stage. The reason for this is that restocking is more expensive if more items are restocked, such that a minimal restock amount appears to be more profitable. Unfortunately, this causes too many restock actions in the long run.

With a fixed value of S, what remains is to find the restock threshold s. Given the inventory level y and demand realization d, the following steps are performed: first, compute the objective value for either performing no restock ($x = 0$) or restocking up to S ($x = S - y$). Second, compute the (weighted) average objective value for each combination of (x, y) with respect to d, as the demand is unknown when deciding the value of x. Lastly, the inventory level y of interest can be extracted; i.e. the smallest inventory level y where performing a restock is more expensive than doing nothing. See Figure 2.5 for an example.

Local Search

The next method which can be performed quite fast as well is to use a local search algorithm to find optimal policy values (s, S). Several different local search algorithms exist, but the main idea is as follows: given some point (s, S), compute the (expected) costs when using these policy values. Next, search in the (local) vicinity for a point (s^*, S^*) with lower (expected) costs. If such a point is found, perform the same steps again at this new point (while iteratively reducing the search space). If no point is found, (s, S) is a (locally) minimal solution. To increase robustness of the local search, initiate from several different start points.

While local search algorithms are computationally fast to execute, it might occur that the algorithm has trouble finding the globally optimal solution, in particular if a lot of local minima exist. Next to that, determining that a minimal solution is in fact the global minimum requires thorough understanding of the behaviour of the objective function.

In our case, the support of the values (s, S) has reasonable size, in particular because we only consider integer values. For example, following the example of Figure 2.5, the inventory level is at most 180. Considering that $S > s$, there are about $16 \cdot 10^3$ values possible, where for each of these values the expected costs can be computed quite fast.[3] The advantage is that the global optimum can be determined exactly, with the downside that a longer computation time is required.

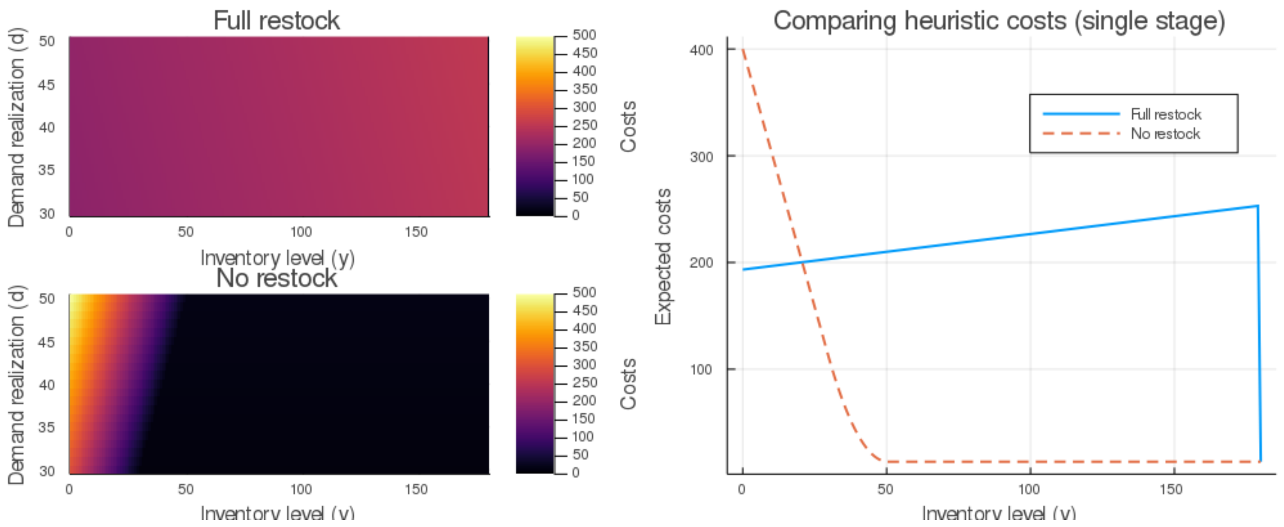


Figure 2.5: An overview of the results from the heuristic policy-finding method. The images on the left provide heatmaps of the costs for specific inventory levels and demand realizations. The image on the right plots the (weighted) average costs for all possible inventory levels.

Stochastic Programming

With the techniques of stochastic programming (see Section 2.2), slight adjustments can be made to said techniques to find (near) optimal policy values (s, S). By constructing a scenario tree and adding constraints based on Equation (2.18), a linear program can be solved, with the aim of choosing optimal policy values (s, S) instead of directly choosing optimal decision variables x. These policy constraints are explained in detail in Section 3.7. When using this approach, note that using the receding time horizon (Section 2.2.1) is not required, as the policy values are used throughout the horizon to determine the decision variables. As such, a much larger scenario tree can be constructed, as it needs to be solved merely once.

Robust Optimization

As described in Section 2.3, a robust optimization can be performed with minor adjustments to the stochastic programming approach. Likewise, similar policy constraints can be added to robust optimization to find the policy values. However, as robust optimization provides very conservative solutions, the policies resulting from it perform poorly.[4] Because of the nature of the inventory control problem, it is much more interesting to perform better on average than to perform optimal in the worst-case scenario. As such, we will not go into detail on further performance of this approach.

[3]It would be possible to decrease the number of possible values even further, considering that s is a relatively small number compared to S. More thought could be put into this if a larger support or several items simultaneously are considered.

[4]This is concluded from initial results. These results are not included in the thesis.

3 | Model Description

The inventory control model we analyse is based on the problem description of Section 1.3, and is solved using the techniques of Chapter 2. As stochastic programming is our main approach, the notation in this chapter will be focused primarily on the stochastic programming method. An overview of the symbols used in this chapter is provided in Table 3.1.

The remainder of this chapter is structured as follows: in Section 3.1, it is explained how the indices are used. Next, the main variables that are used in the model are provided: the demand realizations $\widetilde{D}$ in Section 3.2 and the decision variables in Section 3.3. In Sections 3.4 and 3.5, the constraints and objective function are provided in mathematical terms. Afterwards, the technicalities of using a mixed integer linear program are described (Section 3.6), by providing a list of how several variables and constraints are reformulated. Next, we provide an overview of the adjustments required when using the stochastic program to find the policy values in Section 3.7. Lastly, the complete linear program (as used in the simulation itself) is presented in Section 3.8.

Model Assumptions

Several assumptions and simplifications have been mentioned in previous chapters. Here, an overview of all model assumptions is given.

Restocks and demands are at discrete time events
Instead of simulating a constant stream of orders, we choose to modify the inventory levels only once per day (as an aggregated sum of all orders from that day). Similarly, restock actions are not allowed throughout the day, but only at the start of each day. Because of this, we can simply use time indices t to indicate the restocks and demands per day.

Decisions are made with zero lead time
Whenever a decision is made (i.e. what items to restock), the effects of these decisions are realized immediately. The (time) penalties for restocking and order picking are merely to indicate the relative time required to actually perform such actions. Because of the zero lead time, optimizing which decisions should be made can be done much more accurately.

Perfect state information is used
All state descriptions used (such as current inventory levels and previous demands) are 100% accurate. This way, decisions can be made with a particular state in mind, where there is no ambiguity regarding the current state.

The bulk-pick area has unlimited stock
As mentioned in the introduction (Section 1.3), it is assumed that the bulk-pick area of the warehouse holds enough items in stock. By means of external deliveries, the inventory levels in the bulk-pick area are kept in check, such that we are always able to restock items from this area to the direct-pick area.

All demand is non-negative

The demand is represented by particular orders, and a negative demand would indicate items being brought back to the warehouse. To avoid confusion and unnecessary complexity, the process of returning items is ignored in this model.

Table 3.1: An overview of all parameters and variables used in the model description.

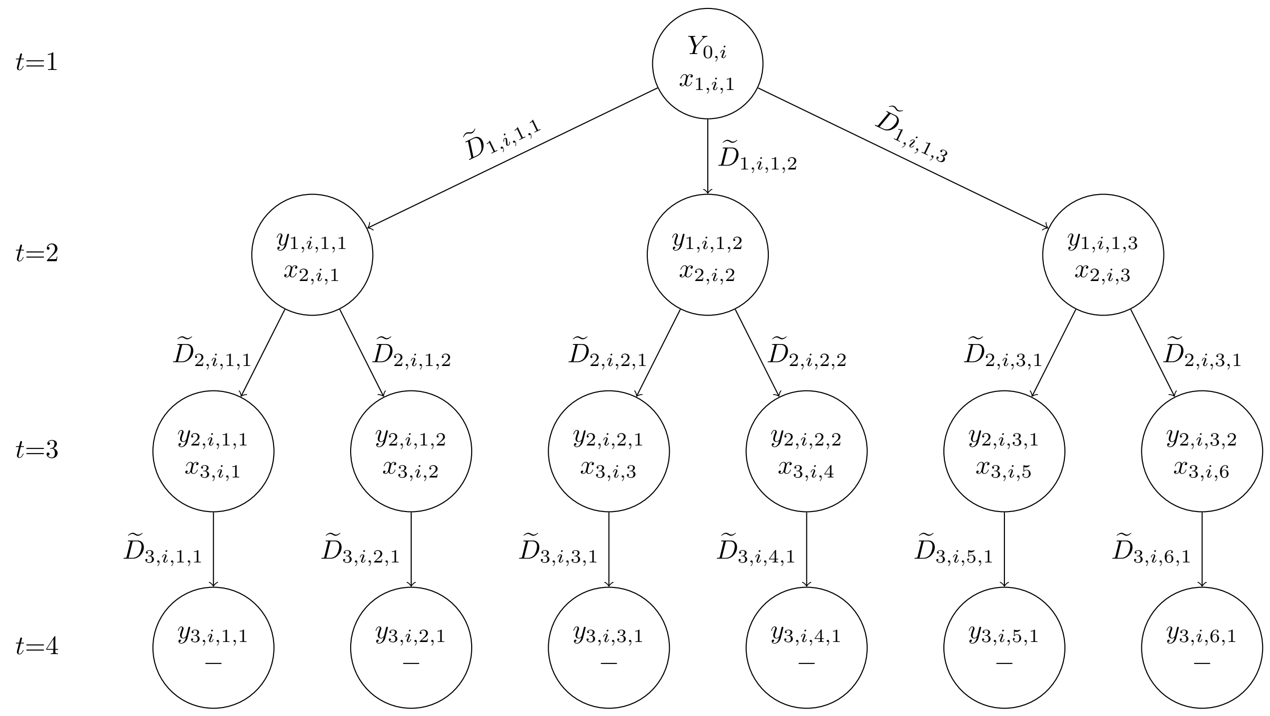
Scenario tree	**Definition**	**Dimension**
T	Depth of tree	1
N	Number of different items (products)	1
S_t	Number of nodes (scenarios) at time t	T
C_t	Number of children per node at time t	T
Indices	**Definition**	**Values**
t	Period number	$\{1,2,...,T\}$
i	Product number	$\{1,2,...,N\}$
j	Scenario number	$\{1,2,...,S_t\}$
k	Child node number	$\{1,2,...,C_t\}$
Decision variables	**Definition**	**Dimension**
x_{tij}	Restock quantity (number of packages)	$T \times N \times S_t$
z_{tij}	Binary decision: is the item restocked?	$\{0,1\}^{T \times N \times S_t}$
σ	Staff level (optional)	1
f_{tj}	Number of flex workers (optional)	$T \times S_t$
Policy variables	**Definition**	**Dimension**
s_i	Restock threshold: restock item i if inventory below s_i	N
S_i	Restock level: restock up to (at least) S_i	N
Implicit variables	**Definition**	**Dimension**
y_{tijk}	Inventory level at the start of day $t+1$	$T \times N \times S_t \times C_t$
$negInv_{tijk}$	How much the inventory level is below 0	$T \times N \times S_t \times C_t$
$restockInv_{tijk}$	The restock inventory amount	$T \times N \times S_t \times C_t$
Uncertainty	**Definition**	**Dimension**
$\widetilde{D}_{tijk}$	Realization / prediction of the demand	$T \times N \times S_t \times C_t$
ρ	Correlation factor (for non-i.i.d. demand)	1
Γ	Pool of all customers (for arrival process)	$\mathbb{N}^+$
Parameters	**Definition**	**Dimension**
P_i	Direct-picking time	N
B_i	Bulk-picking time	N
R_i	Restocking time (constant)	N
A_i	Restocking time per item moved	N
V_i	Maximum inventory space	N
Q_i	Amount of items per container	N
Y_{0i}	Initial inventory level	N
E	Costs per employee (optional)	1
F	Costs per flex worker (optional)	1
L	Labour: number of items each worker can restock (optional)	1
$\mathbb{F}$	Maximum number of flex workers (optional)	1

3.1 Indices

In order to keep track of the decision variables and the inventory levels, we introduce a quadruplet of indices: the indices t, i, j, k indicate the day t, the item number i, the scenario number j and the child node number

k respectively. In Figure 3.1, an example is provided regarding how these indices are used for the three main variables $y, x, \widetilde{D}$. The size of this scenario tree is described by the following parameters:

$$\text{Depth } T = 3 \quad \text{Child nodes } C_t = [3,2,1] \quad \text{Scenarios } S_t := \begin{cases} \prod_{l=1}^{t} C_l & \text{if } t > 1 \\ 1 & \text{else} \end{cases} = [1,3,6,6] \tag{3.1}$$



t=1
t=2
t=3
t=4
$Y_{0,i}$
$x_{1,i,1}$
$\widetilde{D}_{1,i,1,1}$
$\widetilde{D}_{1,i,1,2}$
$\widetilde{D}_{1,i,1,3}$
$y_{1,i,1,1}$ $x_{2,i,1}$
$y_{1,i,1,2}$ $x_{2,i,2}$
$y_{1,i,1,3}$ $x_{2,i,3}$
$\widetilde{D}_{2,i,1,1}$
$\widetilde{D}_{2,i,1,2}$
$\widetilde{D}_{2,i,2,1}$
$\widetilde{D}_{2,i,2,2}$
$\widetilde{D}_{2,i,3,1}$
$\widetilde{D}_{2,i,3,1}$
$y_{2,i,1,1}$ $x_{3,i,1}$
$y_{2,i,1,2}$ $x_{3,i,2}$
$y_{2,i,2,1}$ $x_{3,i,3}$
$y_{2,i,2,2}$ $x_{3,i,4}$
$y_{2,i,3,1}$ $x_{3,i,5}$
$y_{2,i,3,2}$ $x_{3,i,6}$
$\widetilde{D}_{3,i,1,1}$
$\widetilde{D}_{3,i,2,1}$
$\widetilde{D}_{3,i,3,1}$
$\widetilde{D}_{3,i,4,1}$
$\widetilde{D}_{3,i,5,1}$
$\widetilde{D}_{3,i,6,1}$
$y_{3,i,1,1}$ –
$y_{3,i,2,1}$ –
$y_{3,i,3,1}$ –
$y_{3,i,4,1}$ –
$y_{3,i,5,1}$ –
$y_{3,i,6,1}$ –

Figure 3.1: Tree of inventories $y_{t,i,j,k}$, restock decisions $x_{t,i,j}$ and demand realizations $\widetilde{D}_{t,i,j,k}$. Note that i is fixed, and that the indexing of x does not depend on the child node number k.

Index t indicates that the variable is relevant at time t. The exception to this is the inventory variable y, as y_t indicates the inventory level at the start of period $t+1$. The reason for this is that the computation of the inventory level is less cumbersome regarding the indexation. The index i is straightforward, as it indicates the reference number of the item in question.

The remaining two indices j and k attain values depending on the period number t: j indicates the number of the scenario, ranging from $1, 2, ..., S_t$, which is the number of scenarios at period t. In the example Figure 3.1, observe that index j of $x_{3,i,j}$ ranges from 1 to 6, as the number of scenarios at time $t = 3$ is given by $S_3 = C_1 \cdot C_2 = 6$. On a similar note, k indicates the number of the child node with respect to its parent node, ranging from $1, 2, ..., C_t$, which is the total number of child nodes per node when going from depth $t-1$ to t. In the example, the index k of $y_{2,1,j,k}$ attains only the values 1 and 2.

By combining this absolute indexing j and relative indexing k, we can describe the position and relation of all nodes in the scenario tree. To illustrate the usage of the indexing j, k, observe the following: suppose we are at some decision point with indices (t, i, j), and we are interested in the indices j of the scenarios at time $t+1$. It is known that the demand realizations $\widetilde{D}$ have indices (t, i, j, k) where $k \in \{1, 2, ..., C_t\}$. Using this, the indices that follow are given by

$$(t, i, j), \text{ child node } k \Rightarrow (t+1, i, (j-1) \cdot numChilds[t+1] + k) \tag{3.2}$$

Furthermore, for any decision point with indices (t, i, j), we can find the indices of the parent node $(t-1, i, j^*, k^*)$:

$$j^* = \left\lceil \frac{j}{C_{t-1}} \right\rceil \tag{3.3}$$

$$k^* = j - C_{t-1} \cdot (j^* - 1) \tag{3.4}$$

The computation of j^* and k^* can be explained as follows: as the number of child nodes is equal throughout each level of the scenario tree, all scenarios at time $t-1$ have C_{t-1} child nodes. As such, at level t, the first C_t nodes have scenario 1 as parent node, the next C_t nodes scenario 2, and so on. For k^*, we require the difference between j and $C_{t-1} \cdot j^*$, which results in a number modulo C_{t-1}. Note, however, that the numbering of parent nodes starts at 1 instead of 0, such that we subtract 1 from j^*.

Throughout this chapter, we will use j^* and k^* to indicate the scenario- and child number resulting from equations (3.3) and (3.4).

3.2 Sampling Demand

One of the main requirements for the model is that demand (uncertainty) needs to sampled. This demand indicates (per item type) how many products are taken out of stock, either from the direct-pick area or the bulk-pick area. There are several different methods which are used to sample demand; the most straightforward method is to use independent and identically distributed random variables (i.i.d.), where each variable indicates the demand for a single product. The second method is to use random variables which are not i.i.d. distributed, such that a correlation between successive days can be enforced. The third method considered is the simulation of an arrival process of customers, each placing a particular order of products upon arrival. These different demand models are explained in detail in Chapter 4.

Throughout this report, the variable $\widetilde{D}$ is used to indicate a sample of the demand. As such, $\widetilde{D}$ is a (real-valued) number, and never a random variable. This is necessary for the stochastic programming techniques, as these are based on the fact that any randomness and uncertainty is substituted by its samples. For more information on this matter, see Section 2.2.

For the demand realizations $\widetilde{D}$, the only assumption made is that the demand is always non-negative, such that

$$\widetilde{D}_{t,i,j,k} \geq 0 \qquad \forall t, i, j, k \tag{3.5}$$

The reason for this is that a negative demand would indicate an increase in inventory level, for example when products are returned to the warehouse. However, for the sake of this model, this is not taken into account, as mentioned earlier. Next to this assumption, the demand realizations are in general rounded to integer values; while not necessary for the model, it makes sense that a product is never partially demanded (e.g. half a product). As a result, the inventory level y and restock quantity x are also integer-valued.

3.3 Main Variables

Besides the demand variable $\widetilde{D}$, the three variables most used in the model are the restock amount x, the implicit binary restock indicator z and the implicit inventory variable y. Optionally, the decision variables indicating the staffing level σ and the number of flex workers f can be added to the model. In this section, we provide a brief overview of these variables.

3.3.1 Restock Amount x

The main decision variable of the stochastic program is the restock amount $x_{t,i,j}$, which is to be optimized for all investigated periods t, items i and scenarios j. The restock amount indicates the number of containers which are restocked; each container contains Q_i products of type i. As such, the actual amount of products restocked is given by

$$Q_i \cdot x_{t,i,j} \tag{3.6}$$

3.3.2 Binary Restock Indicator z

One of the most commonly used implicit variables used in the model is the restock indicator $z_{t,i,j}$. This variable is used especially for computing the restock costs (Section 3.5.2), as the restock costs are logically 0 if there is no restock. As such, we define $z_{t,i,j}$ as follows:

$$z_{t,i,j} = \mathbb{I}_{\{x_{t,i,j}>0\}} = \begin{cases} 1 & \text{if } x_{t,i,j} > 0 \\ 0 & \text{else} \end{cases} \qquad \forall t,i,j \tag{3.7}$$

3.3.3 Inventory Level y

Another important implicit variable used in the model is the inventory level $y_{t,i,j,k}$. This variable describes the level of the inventory at the *start* of day $t+1$. As the inventory level is influenced by the restock amount x and the realized demand $\widetilde{D}$, the formula is given by

$$y_{t,i,j,k} = \begin{cases} Q_i \cdot x_{t,i,j} + \widetilde{D}_{t,i,j,k} + Y_{0,i} & \text{if } t = 1 \\ Q_i \cdot x_{t,i,j} + \widetilde{D}_{t,i,j,k} + y_{t-1,i,j^*,k^*} & \text{else} \end{cases} \qquad \forall t,i,j,k \tag{3.8}$$

Often, we require the inventory level directly after a restock x, but before the demand $\widetilde{D}$ is subtracted from the inventory level. Using (3.8), observe the equality

$$y_{t,i,j,k} + \widetilde{D}_{t,i,j,k} = x_{t,i,j} + y_{t-1,i,j^*,k^*} \qquad \text{where } y_{0,i,j,k} = Y_{0,i} \qquad \forall t,i,j,k \tag{3.9}$$

When using the left-hand side of (3.9), we do not require to find the previous indices j^* and k^*, which makes the formulation of some constraints easier. See Section 3.6 for more on this matter.

3.3.4 Staffing Level σ and Flex Workers f

An optional module which can be added to the model is the use of employees. Adding employees adds complexity by bounding the number of items that can be restocked, depending on how many employees there currently are. When adding employees, consider that there are two separate types of employees: the fixed staff and the flex workers. The difference between these employees is that the staff level σ is determined and fixed for the entire horizon, while the number of flex workers f can be adjusted on a daily basis. Next to this, the costs for hiring the staff and flex workers are given by E and F respectively, where $E < F$. For convenience, assume that the amount of flex workers can be determined with zero lead time, such that they are determined at the same time as the other decision variables.

Each worker (staff or flex) can restock L different item (types) on a single day, defined as the amount of *labour* of one worker. The total number of workers on a particular moment can be found by taking the sum of the number of staff members σ and the flex workers f. As such, the total number of items that can be restocked on a single day is bounded by

$$L \cdot (\sigma + f_{t,j}) \qquad \forall t,j \tag{3.10}$$

3.4 Constraints

The next ingredient required to use the linear program is to define the list of all constraints used. These are listed in this section.

3.4.1 Maximum Space

The inventory level after a restock may never exceed the maximum amount of space available for that product. For this type of constraint, we require the inventory after the restock x but before the demand $\widetilde{D}$, as formulated in (3.9). Using this formulation, the maximum space constraint is given by

$$y_{t,i,j,k} + \widetilde{D}_{t,i,j,k} \leq V_i \quad \forall t, i, j, k \tag{3.11}$$

3.4.2 Restock up to S

When a restock of item i occurs, we want the inventory level to be restocked up to at least some level S_i (i.e. up to full capacity). However, to decrease the amount of restocking freedom allowed, we also enforce the restock level to be as close as possible above S_i. As such, the inventory level after restocking will be equal to some $y \in \{S_i, S_i + 1, ..., S_i + Q_i - 1\}$. These constraints are combined into a single if-and-only-if statement, as no constraint is required if no restock is performed:

$$z_{t,i,j} = 1 \iff S_i \leq y_{t,i,j,k} + \widetilde{D}_{t,i,j,k} \leq S_i + Q_i - 1 \tag{3.12}$$

Observe that the inventory level after a restock but before the demand is required, such that the reformulation of Equation (3.9) can be used. In addition, note that S_i is either optimized by the model (i.e. by the (s, S) policy), or that it is fixed to indicate a full restock:

$$S_i = V_i - Q_i + 1 \tag{3.13}$$

3.4.3 Maximum Labour by Workers

A described in Section 3.3.4, the total amount of different items that can be restocked is determined by the number of staff members σ and flex workers f. As the total amount of labour that can be performed on a particular day is given by (3.10), the constraint of total restocks that follows is

$$\sum_{i=1}^{N} z_{t,i,j} \leq L \cdot (\sigma + f_{t,j}) \qquad \forall t, j \tag{3.14}$$

3.4.4 Maximum Number of Flex Workers $\mathbb{F}$

It is possible that there are only a limited number of flex workers available, as indicated by $\mathbb{F}$. For example, only a few employees can be hired in addition to the staff, as not everyone is able to restock certain products (i.e. due to heavy weights or dangerous compounds). In particular, it might be that flex workers cannot be hired at all ($\mathbb{F} = 0$), such that all restocks need to be performed by (trained) staff members. As a constraint for the model, we simply add

$$f_{t,j} \leq \mathbb{F} \qquad \forall t, j \tag{3.15}$$

3.5 Objective Function

As a (mixed integer) linear program is used to find the optimal values for the decision variables, a central objective function to be minimized is required. This function consists of several types of costs: restock, direct-pick and bulk-pick, and optionally employee costs. In the sections below, these costs are described in detail.

When computing the objective value, it should be noted that the stochastic program uses a scenario tree, where each scenario has a particular probability of occurring. In the general case, each scenario at time t has a probability of $(S_t)^{-1}$ or $(S_{t+1})^{-1}$ of occurring (depending on whether the variable is dependent on the index

k). As such, when computing the (expected) costs over the entire scenario tree, we take the normalized sum of all costs combined, and aim at minimizing these total costs:

$$\begin{aligned}
\min_x \quad & \text{Staff}(\sigma) + \sum_{t=1}^{T} \frac{1}{S_t} \sum_{j=1}^{S_t} \Bigg[\text{Flex}(t,j) + \sum_{i=1}^{N} \Big[\text{Restock}(t,i,j) \\
& \qquad + \frac{1}{S_{t+1}} \sum_{k=1}^{C_t} \big[\text{Direct-pick}(t,i,j,k) + \text{Bulk-pick}(t,i,j,k) \big] \Big] \Bigg] \\
= \min_x \quad & \sigma \cdot T \cdot E + \sum_{t=1}^{T} \frac{1}{S_t} \sum_{j=1}^{S_t} \Bigg[f_{t,j} \cdot F + \sum_{i=1}^{N} \Big[z_{t,i,j} \cdot (R_i + A_i \cdot (x_{t,i,j} + 2 \cdot y_{t-1,i,j^*,k^*} + \max(0, -y_{t-1,i,j^*,k^*}))) \\
& \qquad + \frac{1}{S_{t+1}} \sum_{k=1}^{C_t} \Big[P_i \cdot \Big(\widetilde{D}_{t,i,j,k} - \max(0, -y_{t,i,j,k}) \Big) + B_i \cdot \max(0, -y_{t,i,j,k}) \Big] \Big] \Bigg] \qquad (3.16)
\end{aligned}$$

3.5.1 Employee Costs

Employees are an optional module which can be added to the model (i.e. if the costs $E = F = 0$ are fixed, setting employee levels is irrelevant for the objective value). The employee costs consist of two separate cost functions: the staff costs and the flex costs. As the number of staff members is fixed over the horizon, the staff costs can be taken out of the sum completely. The costs per staff member is E per time step, such that the total staff costs is given by

$$\text{Staff}(\sigma) = \sigma \cdot T \cdot E \qquad (3.17)$$

If there are not enough staff members to restock all items on a particular day, additional flex workers can be hired. The costs per flex workers is F at any time step, and as the number of flex workers can vary per scenario, the costs are given by

$$\text{Flex}(t,j) = f_{t,j} \cdot F \qquad (3.18)$$

3.5.2 Restock Costs

Restock actions are performed in order to increase the inventory level of the direct-pick area. One must note that, when restocking this area, it is important that products are placed according to a First-In-First-Out policy (FIFO). The reason for this is that products might have a limited shelf life. Using this property, it can be seen that the restock costs consist of two main aspects: a constant time R_i required to restock, and a time-per-product A_i needed to physically move the products.

Suppose that the inventory level in the direct-pick area is equal to y products, and we wish to restock x items. Then the time required to perform a restock is equal to a fixed time R_i (for going to and returning from the bulk-pick area to retrieve the products), then removing y items from the picking area (using the FIFO policy), putting x items on the shelf, and finally placing the y taken out items back in the direct-pick area (by placing them in front of all other items). In terms of a formula, this boils down to

$$\text{Restock}(t,i,j) = z_{t,i,j} \cdot (R_i + A_i \cdot (x_{t,i,j} + 2 \cdot y_{t-1,i,j^*,k^*} + \max(0, -y_{t-1,i,j^*,k^*}))) \qquad (3.19)$$

Note that, for a restock during period t, we require the inventory variable at period $t-1$, as this indicates the number of items in stock at the time of restocking. In addition, the inventory level y can be 'negative' at some point (i.e. a bulk-pick was needed in the previous period), a restock might occur where there are no products in the direct-pick area. Below, we show that (3.19) incorporates all possible inventory levels.

Positive inventory

Let $y_{t-1,i,j^*,k^*} \geq 0$. Then the restock amount $x_{t,i,j}$ and current inventory level y_{t-1,i,j^*,k^*} are correct. Using the fact that $\max(0, -y_{t-1,i,j^*,k^*}) = 0$, we get

$$z_{t,i,j} \cdot (R_i + 2 \cdot A_i \cdot y_{t-1,i,j^*,k^*} + A_i \cdot x_{t,i,j}) \tag{3.20}$$

Negative inventory

Let $y_{t-1,i,j^*,k^*} < 0$. Then the *actual* restock amount is $-y_{t-1,i,j^*,k^*}$ higher than $x_{t,i,j}$, as we first 'restock' up to 0 inventory, and afterwards actually restock the products. To compensate for this, we add y to x. Note that, as we need to FIFO fill 0 items, and $\max(0, -y_{t-1,i,j^*,k^*}) = -y_{t-1,i,j^*,k^*}$, we obtain

$$z_{t,i,j} \cdot (R_i + A_i \cdot y_{t-1,i,j^*,k^*} + A_i \cdot x) \tag{3.21}$$

3.5.3 Bulk-Pick Costs

The bulk-picking costs are relatively high, but only occur if the direct-picking area does not contain enough products. If the inventory level $y_{t,i,j,k}$ is negative, it means that there were not enough items in the direct-pick area (after restocking) to fulfil the full demand $\widetilde{D}_{t,i,j,k}$. For each of these items, a bulk-pick time B_i is needed.[1] As such, the bulk-pick costs are given by

$$\text{Bulk-pick}(t, i, j, k) = B_i \cdot \max(-y_{t,i,j,k}, 0) \tag{3.22}$$

3.5.4 Direct-Pick Costs

The direct-picking costs cannot be prevented, as these describe the process of order-picking and thus satisfying the demand of each day. Per product of type i taken out of the direct-picking area, a time P_i is required. The amount of products taken from the direct-pick area is equal to $\widetilde{D}_{t,i,j,k}$, except when there are not enough items in stock. If this occurs, a bulk-pick time is required instead of a direct-pick time. By compensating for this, the direct-pick costs are given by

$$\text{Direct-pick}(t, i, j, k) = P_i \cdot \left(\widetilde{D}_{t,i,j,k} - \max(y_{t,i,j,k}, 0)\right) \tag{3.23}$$

3.6 Reformulation of Variables and Constraints

Some of the current formulations of variables and constraints cannot be used in a linear program; in particular, if-else statements are not supported directly. As such, it is required that we rewrite these formulations, in particular by using big-M constraints, such that they can be used. The basic idea of a big-M constraint it that an upper bound is constructed by multiplying a large number M with some binary variable. This way, if the binary variable is equal to 1, the constraint is always satisfied by the fact that M is an upper bound.

3.6.1 Reformulating $z_{t,i,j}$

As the formulation of $z_{t,i,j}$ in Equation (3.7) is based on two if-and-only-if statements, it needs to be rewritten it using two separate big-M constraints:

$$z_{t,i,j} \leq x_{t,i,j} \qquad \forall t, i, j \tag{3.24}$$

$$x_{t,i,j} \leq M \cdot z_{t,i,j} \quad \forall t, i, j \tag{3.25}$$

[1]These costs are not entirely realistic, as the bulk-pick time should have a similar structure as the restock costs. However, this further increases the complexity of the model, while not having a significant impact on the rest of the model.

If $x_{t,i,j} = 0$, then by (3.24) it follows that $z_{t,i,j} = 0$, and (3.25) is satisfied. However, if $x_{t,i,j} > 0$, then $z_{t,i,j} = 1$ in order to satisfy (3.25). We wish to choose M as a small upper bound to x: note that the number of items restocked can never exceed V_i (the maximum inventory space) plus the lowest attainable inventory level $y_{t,i,j,k}$. We assume $y_{t,i,j,k} > -V_i$, such that

$$M = \max_i \left(\frac{2 \cdot V_i}{Q_i} \right) \tag{3.26}$$

3.6.2 Reformulating the Restock Constraint

As described in Section 3.4.2, a constraint is added indicating that the restock level should be at least equal to S_i. As this constraint is an if-and-only if statement, it can be rewritten using two big-M constraints:

$$\begin{aligned} y_{t,i,j,k} + \widetilde{D}_{t,i,j,k} + (1 - z_{t,i,j}) \cdot M \geq S_i \quad \forall t, i, j, k \\ y_{t,i,j,k} + \widetilde{D}_{t,i,j,k} + (1 - z_{t,i,j}) \cdot M \leq S_i + Q_i - 1 \quad \forall t, i, j, k \end{aligned} \tag{3.27}$$

Again, an upper bound to $y_{t,i,j,k} + \widetilde{D}_{t,i,j,k}$ for the value of M is required. As a rough upper bound, we use $M = 2 \cdot V_i$.

3.6.3 Reformulating $\max(0, -y_{t,i,j,k})$

If the inventory level of the direct-pick area becomes negative, it means that not all demand of the previous period could be picked from the direct-pick area. As such, a bulk-pick time penalty is required. To use a max function in a linear program, we define a dummy variable $negInv_{t,i,j,k}$, given by

$$negInv_{t,i,j,k} \geq 0 \qquad \forall t, i, j, k \tag{3.28}$$

$$negInv_{t,i,j,k} \geq -y_{t,i,j,k} \quad \forall t, i, j, k \tag{3.29}$$

The reason why we can utilize such a rewrite is because our objective function is a minimization function. As such, the solver will choose each *negInv* variable variable to be a value which is as small as possible, while still satisfying both constraints. In short, *negInv* is equal to the maximum of both 0 and $-y$.

3.6.4 Reformulating Restock(t, i, j)

The restock costs function (Section 3.5.2) contains quite a few difficulties: it uses both the binary restock variable z, as well as the dummy variable *negInv*. In addition, observe that (3.19) contains a multiplication of z and y. As we wish to use a linear program, these quadratic terms need to be reformulated. As such, we define a new dummy variable *restockInv*, which is defined as

$$restockInv_{t-1,i,j^*,k^*} = z_{t,i,j} \cdot (2 \cdot y_{t-1,i,j^*,k^*} + negInv_{t-1,i,j^*,k^*}) \tag{3.30}$$

Unfortunately, we cannot define a variable by means of a quadratic variable constraint. To circumvent this, four constraints are required: let L and U be the lower and upper bound of $(2 \cdot y + negInv)$. Then we rewrite Equation (3.30) to

$$\begin{aligned} restockInv_{t-1,i,j^*,k^*} &\leq U \cdot z_{t,i,j} \quad \forall t, i, j \\ restockInv_{t-1,i,j^*,k^*} &\geq L \cdot z_{t,i,j} \quad \forall t, i, j \\ restockInv_{t-1,i,j^*,k^*} &\leq (2 \cdot y_{t-1,i,j^*,k^*} + negInv_{t-1,i,j^*,k^*}) - L \cdot (1 - z_{t,i,j}) \quad \forall t, i, j \\ restockInv_{t-1,i,j^*,k^*} &\geq (2 \cdot y_{t-1,i,j^*,k^*} + negInv_{t-1,i,j^*,k^*}) - U \cdot (1 - z_{t,i,j}) \quad \forall t, i, j \end{aligned} \tag{3.31}$$

The equations above can most easily be verified by showing that they hold for both $z = 0$ and $z = 1$: if $z = 0$, then the first two equations combine to $restockInv = 0$, and the latter two are always valid because of L and U. If $z = 1$, the reasoning is almost identical, except that we conclude $restockInv = 2 \cdot y + negInv$ by the last two equations.

3.7 Policy Adjustments

It is possible to use the technique of stochastic programming to determine (base stock) policy values. When we wish to do this, a few constraints and variables need to be added to the linear program, as described below.

3.7.1 Base Stock Policy Optimization

When using stochastic programming to find base stock policy values (as described in Section 2.4), it is required to make a few adjustments to the linear program. Instead of directly optimizing the decision variables x, we instead optimize the policy values (s_i, S_i) for all $i \in \{1, ..., N\}$, which fully describe what the value of $x_{t,i,j}$ is at every step: if the inventory level y_{t-1,i,j^*,k^*} drops below s_i, we restock up to (at least) S_i. If not, we do not restock item i. This can be described using two simple if-and-only-if statements:

$$\begin{aligned} y_{t-1,i,j^*,k^*} < s_i \quad &\iff \quad z_{t,i,j} = 1 \qquad \forall t, i, j \\ y_{t-1,i,j^*,k^*} \geq s_i \quad &\iff \quad z_{t,i,j} = 0 \qquad \forall t, i, j \end{aligned} \tag{3.32}$$

These if-and-only-if constraints can be added to the linear program using big-M constraints. Below, the big-M formulation for the above equations is given.

$$y_{t-1,i,j^*,k^*} < s_i + M \cdot (1 - z_{t,i,j}) \qquad \forall t, i, j \tag{3.33}$$
$$y_{t-1,i,j^*,k^*} \geq s_i - M \cdot z_{t,i,j} \qquad \forall t, i, j \tag{3.34}$$

By case distinction, we can check that the big-M formulation indeed satisfies the original if-and-only-if constraints:

$$\begin{aligned} y_{t-1,i,j^*,k^*} < s_i \implies &\begin{cases} (3.33) & \checkmark \\ (3.34) & \implies z_{t,i,j} = 1 \end{cases} \\ y_{t-1,i,j^*,k^*} \geq s_i \implies &\begin{cases} (3.33) & \implies z_{t,i,j} = 0 \\ (3.34) & \checkmark \end{cases} \\ z = 0 \implies &\begin{cases} (3.33) & \checkmark \\ (3.34) & \implies y_{t-1,i,j^*,k^*} \geq s_i \end{cases} \\ z = 1 \implies &\begin{cases} (3.33) & \implies y_{t-1,i,j^*,k^*} < s_i \\ (3.34) & \checkmark \end{cases} \end{aligned} \tag{3.35}$$

Furthermore, observe that the efficiency of the model solving can be increased by indicating possible values of s_i. As a lower bound, s_i should never be less than 0, as then we will always pay backorder costs. As an upper bound, s_i cannot be larger than S_i: this would result in always performing a restock, regardless of what the demand or inventory level is. As such, we add the constraints

$$0 \leq s_i \leq S_i \qquad \forall i \tag{3.36}$$

3.7.2 Determining Flex Workers

When incorporating the employment module in the base stock policy, a few adjustments need to be made. Initially, instead of merely determining the optimal policy values (s_i, S_i) during initialization, the staffing level σ needs to be determined as well. With these (policy) values determined, the only decision which needs to be made is that of the flex workers f.

Given some inventory level $y = (y_1, y_2, ..., y_N)$, it can be determined which restocks z_i are performed, using (3.32). When determining the number of flex workers, we distinguish between three separate cases:

No flex workers are needed

Set $f = 0$ if there are enough staff members to restock all items on that day.

Hire flex workers to restock all items

By setting $f \leq \mathbb{F}$, the total labour on that day is enough to restock all items.

Not enough flex workers can be hired

Even when setting $f = \mathbb{F}$, not all items can be restocked.

The first two cases do not cause any difficulties when determining the number of flex workers and which items to restock (i.e. all as indicated by the base stock policy). In the third case, however, a decision has to be made considering which items to restock. This last case is explained in detail below.

Suppose that, with $f = \mathbb{F}$, a total of k items can be restocked, while the total number of items which needs to be restocked is given by n. Then there are a total of $\binom{n}{k}$ options of which items are restocked. In order to determine which items are restocked, several different approaches can be made:

Random selection

Randomly select k of the n items to restock. No additional information or decision making is required, however this approach might perform far from optimal.

Lowest inventory level

Select the k items with the lowest inventory level. By ordering the items by inventory level, these can easily be found without any difficult decisions. Unfortunately, not all inventory levels might decrease at similar speeds.

Lowest inventory level after realizing demand

As an addition to the previous approach, we can determine the new inventory level by subtracting the (expected / simulated) demand, and *then* choosing to restock the k items with the lowest new inventory level. While this does capture the demand, it does not take into account that not all products have similar bulkpick and restock costs.

Determine the costs for all possible cases

For each of the $\binom{n}{k}$ possible restock situations, the costs can be determined. With all these costs determined, picking the optimal restock situation is trivial. However, determining all $\binom{n}{k}$ costs might require some computation time.

From the aforementioned approaches, the latter gives the best restock solution, and remains computationally feasible as long as $\binom{n}{k}$ is "small". For the computation of the costs, it is merely needed to know all parameter values and the expected demand $\mathbb{E}[D_i]$ for each item i. By choosing a subset of items to restock $\mathscr{R}$, with $|\mathscr{R}| = k$, the costs are given by

$$\text{Costs when restocking } \mathscr{R} \to \sum_{i \in \mathscr{R}} \left[R_i + A_i \cdot (S_i + y_i = \max(0, -y_i))\right] + \sum_{i \notin \mathscr{R}} \left[B_i \cdot \max(0, y_i - \mathbb{E}[D_i])\right] \tag{3.37}$$

3.8 Mixed Integer Linear Program

Using the constraints, variable reformulations and the objective function described in this chapter, we can formulate the entire (mixed integer) linear program used.

$$\begin{aligned}\min_{x} \quad \sigma \cdot T \cdot E + \sum_{t=1}^{T} \frac{1}{S_t} \sum_{j=1}^{S_t} \Bigg[f_{t,j} \cdot F + \sum_{i=1}^{N} \Big[z_{t,i,j} \cdot (R_i + A_i \cdot (x_{t,i,j} + 2 \cdot y_{t-1,i,j^*,k^*} + \max(0, -y_{t-1,i,j^*,k^*}))) \\ + \frac{1}{S_{t+1}} \sum_{k=1}^{C_t} \left[P_i \cdot \left(\widetilde{D}_{t,i,j,k} - \max(0, -y_{t,i,j,k}) \right) + B_i \cdot \max(0, -y_{t,i,j,k}) \right] \Big] \Bigg] \end{aligned} \tag{3.38}$$

subject to

$$y_{t,i,j,k} = Q_i \cdot x_{t,i,j} - \widetilde{D}_{t,i,j,k} + y_{t-1,i,j^*,k^*} \quad \forall t,i,j,k \tag{3.39}$$
$$y_{t,i,j,k} \leq V_i - \widetilde{D}_{t,i,j,k} \quad \forall t,i,j,k \tag{3.40}$$
$$z_{t,i,j} \leq x_{t,i,j} \quad \forall t,i,j \tag{3.41}$$
$$x_{t,i,j} \leq M \cdot z_{t,i,j} \quad \forall t,i,j \tag{3.42}$$
$$negInv_{t,i,j,k} \geq 0 \quad \forall t,i,j,k \tag{3.43}$$
$$negInv_{t,i,j,k} \geq -y_{t,i,j,k} \quad \forall t,i,j,k \tag{3.44}$$
$$y_{t,i,j,k} \geq S_i - \widetilde{D}_{t,i,j,k} - (1 - z_{t,i,j}) \cdot M \quad \forall t,i,j,k \tag{3.45}$$
$$y_{t,i,j,k} \leq S_i + Q_i - 1 - \widetilde{D}_{t,i,j,k} - (1 - z_{t,i,j}) \cdot M \quad \forall t,i,j,k \tag{3.46}$$
$$x_{t,i,j} \geq 0 \quad \forall t,i,j \tag{3.47}$$
$$x_{t,i,j} \text{ integer} \quad \forall t,i,j \tag{3.48}$$
$$z_{t,i,j} \text{ binary} \quad \forall t,i,j \tag{3.49}$$

If the employees module is used, add:

$$L \cdot (\sigma + f_{t,j}) \leq \sum_{i=1}^{N} z_{t,i,j} \quad \forall t,j \tag{3.50}$$
$$\sigma \text{ integer} \tag{3.51}$$
$$f_{t,j} \text{ integer} \quad \forall t,j \tag{3.52}$$

If the base stock policy values are requested, add:

$$y_{t-1,i,j^*,k^*} < s_i + M \cdot (1 - z_{t,i,j}) \quad \forall t,i,j,k \tag{3.53}$$
$$y_{t-1,i,j^*,k^*} \geq s_i - M \cdot z_{t,i,j} \quad \forall t,i,j,k \tag{3.54}$$
$$0 \leq s_i \leq S_i \quad \forall i \tag{3.55}$$
$$s_i \text{ integer} \quad \forall i \tag{3.56}$$

4 | Demand Models

As explained in Section 3.2, sampling demand is an important aspect of the model. There are several different methods in order to create such demand realizations: the straightforward method is to construct a random D_i per item i, such that the demand on a particular day is given by realizations of all D_i. In this method, a distinction can be made between independent or dependent samples. A different method is based on using an arrival process of customers, such that the orders of customers provide the daily demand. These methods are explained in detail in the remainder of this chapter.

4.1 Independent and Identically Distributed Demand

The first demand model described is the sampling of independent and identically distributed (i.i.d.) random variables D_i for $i = 1, ..., N$, where each D_i indicates the demand for item i. As the random variables are i.i.d., no time index t or scenario index j are required. In fact, each of the S_t demand samples needed at time t in the scenario tree are easily sampled as i.i.d. copies.

The advantages of using i.i.d. demand distributions is that it is the easiest and most straightforward method of sampling demand values: only a single distribution is required per item, and no information on past demands needs to be stored. Next to that, several analytical results can be proven when the uncertainty is independent of the past (as seen in Section 2.1). However, it is often not a realistic approach, as uncertainty is almost never independent of the past.

4.2 Dependent Demand

The second, more complicated demand model used is sampling dependent yet identically distributed (non-i.i.d.) random variables. The main idea is that a correlation factor $\rho \in [-1, 1]$ can be specified, such that successive demand realizations have a correlation of around ρ. Two demand realizations $\widetilde{D}_t, \widetilde{D}_{t+1}$ are successive if they share a common scenario j at time t; in other words, if they are child nodes of the same parent scenario. In general, it holds that

$$\widetilde{D}_{t+1,i,j,k} \xRightarrow{\text{succeeds}} \widetilde{D}_{t,i,j^*,k^*} \qquad \forall t, i, j, k \tag{4.1}$$

Note that k^* only depends on j and not on k, such that each $\widetilde{D}_{t,i,j,k}$ is succeeded by $numChilds[t+1]$ demand realizations.

In order to create such a correlation between successive demand realizations, a method is needed to enforce this dependency. Unfortunately, the common method of sampling correlated random variables is by using a *copula* (e.g. a Gaussian copula). However, these generate a pair of correlated random variables, where we require a *new* value given the previous realization. As such, several methods of sampling a new value, conditioned on a correlation factor, have been investigated. Details of these methods are provided in the following sections.

The advantage of using non-i.i.d. demand with respect to the simpler i.i.d. demand (Section 4.1) is that an (anti)-correlation of successive demands often makes sense: if the demand of a particular item is relatively high today, it is probably lower tomorrow. However, one must be careful that the correlated new samples are taken from the same distribution; if not, new samples might become too much distorted from the original distribution for longer sampling horizons. Next to this, the demand only influences samples directly succeeding it, such that there is still a strong Markovian property to the uncertainty.

4.2.1 Uniform Number Sampling

Several of the investigated methods are based on the *inverse transform sampling* technique for random variables. It is based on the following: generate random numbers between 0 and 1, and treat these as probabilities. Then, given some random variable X, the probability density function of X can be constructed by taking the quantile function (or inverse cumulative distribution function) $F^{-1}(\cdot)$ of the uniform numbers. For our case, we perform the following steps of translating to and from uniform numbers:

1. Let $\widetilde{D} \sim D$ be the previous demand realization, with D an arbitrary random variable
2. Define $u_1 := F_D(\widetilde{D})$, where $F_D(\cdot)$ is the cumulative distribution function of D. Note that $F_D(D) \sim \mathcal{U}[0,1]$
3. Generate a new number $u_2 \in [0,1]$, using u_1 and the correlation factor ρ
4. Return the new demand realization $\widetilde{D}^* := F_D^{-1}(u_2)$

Notice that both $\widetilde{D}$ and $\widetilde{D}^*$ are based on the demand distribution D, but have a correlation of approximately ρ by construction. The exact distribution of $\widetilde{D}^*$ and correlation factor ρ depend heavily on the third step, for which a few methods have been investigated. These methods are described in detail below.

Adding Uniform Noise

The first method of generating a correlated number u_2 is by adding "noise" to the previous realization u_1. This noise is defined as a sample of a uniform random variable. Based on the following restrictions, we build a function $Y_1(u_1, \rho) : [0,1] \to [0,1]$:

$$Y_1(u_1, 1) = u_1 \tag{4.2}$$
$$Y_1(u_1, 0) = \mathcal{U}[0,1] \tag{4.3}$$
$$Y_1(u_1, -1) = 1 - u_1 \tag{4.4}$$

By construction, Y_1 is a function representing a random variable. For all other values of ρ, a linear combination of the noise and the previous realization u_1 is needed. As such, define Y_1 as follows:

$$Y_1(u_1, \rho) := \begin{cases} \rho \cdot u_1 + \mathcal{U}[1-\rho] & \text{for } \rho \in [0,1] \\ \rho \cdot u_1 + \mathcal{U}[1+\rho] - \rho & \text{for } \rho \in [-1,0] \end{cases} \tag{4.5}$$

Figure 4.1 displays possible values that $Y_1(u_1, \rho)$ can attain. Multiple lines in each plot indicate (a sample of) possible values. For example, if $(u_1, \rho) = (0.25, 0.5)$, then Y_1 attains values uniformly on $[0.125, 0.625]$.

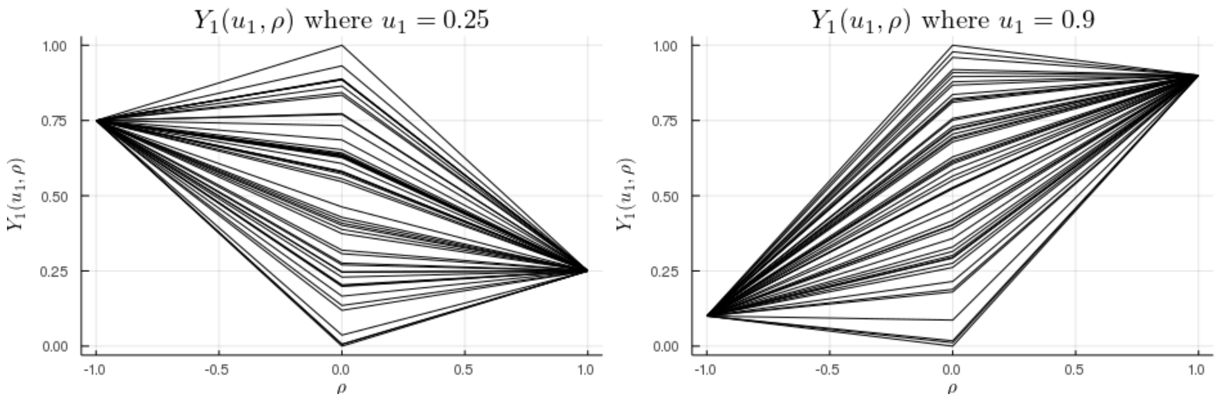


Figure 4.1: Examples of possible values of Y_1, plotted for fixed u_1 and varying correlation factors ρ.

As the goal is to have a correlation of ρ between $\widetilde{D}$ and $\widetilde{D}^*$, we compute the correlation between $u1$ and $Y_1(u_1,\rho)$. These correlation factors are not exactly equal to one another, but could be computed using the following theorem: [Cuadras, 2002]

Theorem 1. *Let X and Y be random variables, and $\alpha(\cdot)$ and $\beta(\cdot)$ functions defined on $[a,b]$ and $[c,d]$ respectively. Assume that both functions have bounded variation and bounded expectational values. Then*

$$cov(\alpha(X),\beta(Y)) = \int_a^b \int_c^d (F_{X,Y}(x,y) - F_X(x)\cdot F_Y(y))\cdot d\alpha(X)\cdot d\beta(Y) \tag{4.6}$$

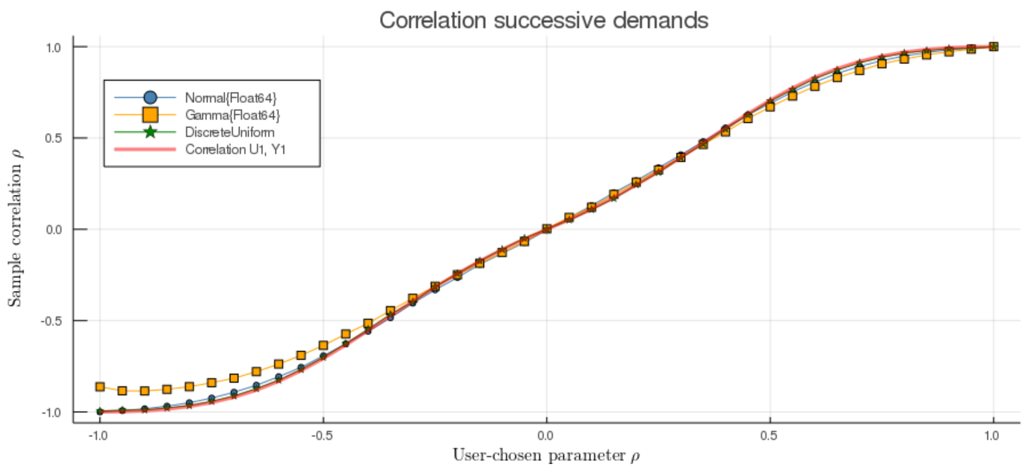


Figure 4.2: Comparison between the correlation factors of the normalized variables and the actual variables.

In our case, we set $X = \widetilde{D}, Y = \widehat{D}^*$, and $\alpha(\cdot) \equiv \beta(\cdot) = F_D(\cdot)$ on the interval $[0,1]$. However, the marginal cumulative distribution function $F_{\widetilde{D}^*}(\cdot)$ is quite difficult to compute, such that we omit the usage of this theorem. Instead, we assume that the correlation between $\widetilde{D}$ and $\widetilde{D}^*$ is similar to the factor between $u1$ and $Y_1(u_1,\rho)$. As such, we only need to find the correlation factor of u_1 and $Y_1(u_1,\rho)$. In order to determine this correlation factor, define $U_\rho := \mathcal{U}[0,1-|\rho|]$ and $U_1 := \mathcal{U}[0,1]$, such that u_1 is a realization of U_1. For notation, let $Y_1 := Y_1(u_1,\rho)$.

Then the correlation is computed as follows:

$$
\begin{aligned}
\mathbb{E}[Y_1 \mid U_1] &= \begin{cases} \mathbb{E}[\rho \cdot U_1 - \rho + U_\rho] & \text{for } \rho \in [-1,0] \\ \mathbb{E}[\rho \cdot U_1 + U_\rho] & \text{for } \rho \in [0,+1] \end{cases} \\
&= \begin{cases} \frac{\rho}{2} - \rho + \frac{1+\rho}{2} & \text{for } \rho \in [-1,0] \\ \frac{\rho}{2} + \frac{1-\rho}{2} & \text{for } \rho \in [0,+1] \end{cases} \\
&= \frac{1}{2} && (4.7) \\
cov(U_1, Y_1) &= \mathbb{E}\left[cov(U_1, Y_1 \mid U_1)\right] + cov\left(\mathbb{E}[U_1 \mid U_1], \mathbb{E}[Y_1 \mid U_1]\right) && \triangleright \text{ Law of total covariance} \\
&= \begin{cases} \mathbb{E}\left[cov(U_1, \rho \cdot U_1 - \rho + U_\rho) + cov(U_1, \frac{1}{2})\right] & \text{for } \rho \in [-1,0] \\ \mathbb{E}\left[cov(U_1, \rho \cdot U_1 + U_\rho) + cov(U_1, \frac{1}{2})\right] & \text{for } \rho \in [0,+1] \end{cases} && \triangleright \text{ Using (4.7)} \\
&= \mathbb{E}[cov(U_1, \rho \cdot U_1) + cov(U_1, U_\rho) && \triangleright\ cov(X, \text{constant}) \equiv 0 \\
&= \frac{\rho}{12} && \triangleright\ U_\rho \text{ is independent} \quad (4.8) \\
\mathbb{V}[Y_1] &= \mathbb{E}[\mathbb{V}[Y_1 \mid U_1]] + \mathbb{V}[\mathbb{E}[Y_1 \mid U_1]] && \triangleright \text{ Law of total variance} \\
&= \begin{cases} \mathbb{V}[\rho \cdot U_1 - \rho + U_\rho] + \mathbb{V}[\frac{1}{2}] & \text{for } \rho \in [-1,0] \\ \mathbb{V}[\rho \cdot U_1 + U_\rho] + \mathbb{V}[\frac{1}{2}] & \text{for } \rho \in [0,+1] \end{cases} \\
&= \mathbb{V}[\rho \cdot U_1] + \mathbb{V}[U_\rho] && \triangleright \text{ All covariances are 0} \\
&= \frac{\rho^2}{12} + \frac{(1-|\rho|)^2}{12} && (4.9) \\
corr(U_1, Y_1) &= \frac{cov(U_1, Y_1)}{\sigma(U_1) \cdot \sigma(Y_1)} \\
&= \frac{\rho/12}{\sqrt{\rho/12} \cdot \sqrt{(\rho^2 + (1-|\rho|)^2)/12}} \\
&= \frac{\rho}{\sqrt{\rho^2 + (1-|\rho|)^2}} && (4.10)
\end{aligned}
$$

This theoretical correlation factor is in line with simulation results. Next, we check our prior statement about the similarity between this correlation factor and the true correlation between $\widetilde{D}$ and $\widetilde{D}^*$. This similarity is plotted for several demand distributions in Figure 4.2. Of those investigated, only the Gamma distribution shows significant deviation near $\rho = -1$ from the theoretical results.

Sampling using Equality Probability

A different method used to enforce the correlation exactly is by defining a random function Y_2, which is equal to its input with a specific probability:

$$\rho \in [-1,0] \implies Y_2(U_1, \rho^-) := \begin{cases} 1 - U_1 & \text{with } \mathbb{P} = |\rho| \\ \mathcal{U}[0,1] & \text{with } \mathbb{P} = 1 - |\rho| \end{cases} \tag{4.11}$$

$$\rho \in [0,+1] \implies Y_2(u_1, \rho^+) := \begin{cases} U_1 & \text{with } \mathbb{P} = \rho \\ \mathcal{U}[0,1] & \text{with } \mathbb{P} = 1 - \rho \end{cases} \tag{4.12}$$

As the random variable Y_2 attains values of two separate random variables with particular probabilities, it can be seen that

$$\begin{aligned} corr(U_1, Y_2) &= \begin{cases} corr(U_1, 1 - U_1) & \text{with } \mathbb{P} = |\rho| \text{ if } \rho \in [-1, 0] \\ corr(U_1, U_1) & \text{with } \mathbb{P} = |\rho| \text{ if } \rho \in [0, +1] \\ corr(U_1, \mathcal{U}[0, 1]) & \text{with } \mathbb{P} = 1 - |\rho| \end{cases} \\ &= \begin{cases} -1 & \text{with } \mathbb{P} = |\rho| \text{ if } \rho \in [-1, 0] \\ 1 & \text{with } \mathbb{P} = |\rho| \text{ if } \rho \in [0, +1] \\ 0 & \text{with } \mathbb{P} = 1 - |\rho| \end{cases} \end{aligned} \tag{4.13}$$

However, by the construction of Y_2, the distribution of the new demand D_2 is nothing like the original D. More on this in the next section.

Comparing the Uniform Samplers

The uniform samplers Y_1 and Y_2 described above can both be used to generate correlated uniform samples. For Y_1, the actual correlation of Y_1 with U_1 is not equal to the user-defined correlation factor ρ, as seen in (4.10). This could be prevented by determining the inverse function, but this is outside the scope of this research. Regardless, the method of adding uniform noise retains parts of the original distribution, as seen clearly in Figure 4.3: the distribution of Y_1 (given one particular value $\widetilde{D}$) is uniform on a smaller interval between 0 and 1.

For Y_2, however, the true correlation factor matches the user-defined factor exactly. The downside, unfortunately, is that the behaviour of the resulting demand D_2 is undesirable: it provides far too many samples which are equal to the previous. For example, if we generate 4 new demand samples, using a user-chosen correlation factor of $|\rho| = 0.6$, then the probability that all of these new samples are the same is $\rho^4 \approx 0.13$. If this would occur when generating new samples in a scenario tree, there is no advantage to investigating more scenarios, as each of these new scenarios is the same. Next to that, the distribution of $\widetilde{D}^*$ is significantly different than the original demand distribution. See Figure 4.3 for a visual representation of this behaviour.

As the distribution of the new demand is much more important than the actual correlation between successive demands, we will primarily use the method of adding uniform noise to generate new correlated uniform samples.

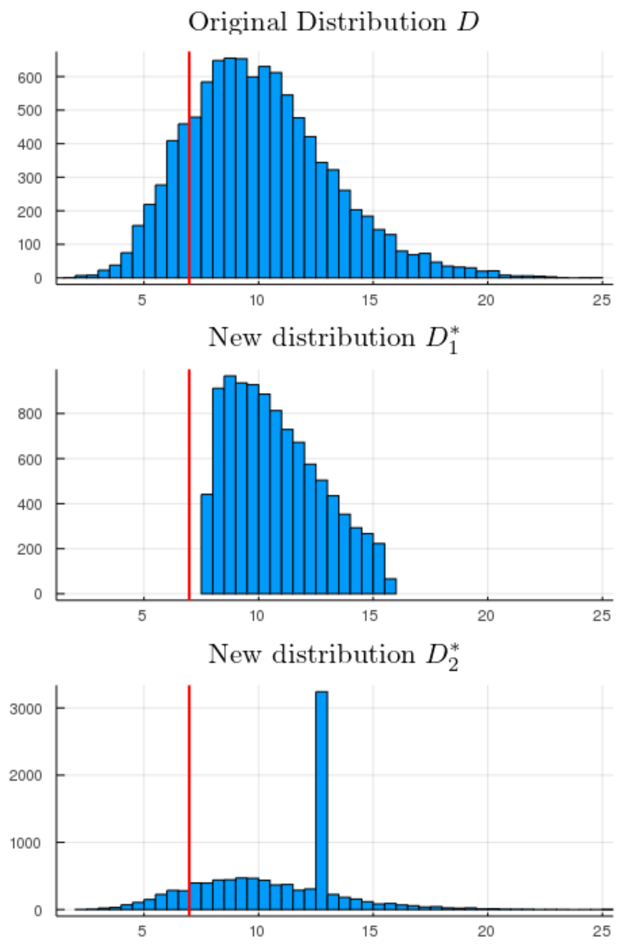
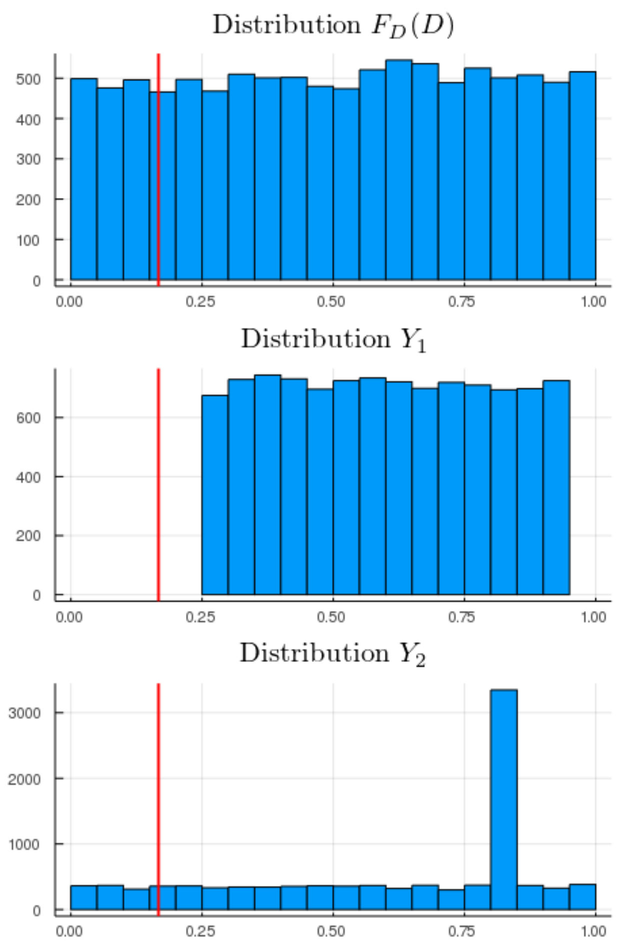


Figure 4.3: Comparison of the discussed correlated uniform number generators Y_1 and Y_2 (Section 4.2.1). The images on the left show the distribution of the actual demand, where the red line indicates a realization $\widetilde{D}$. On the right, the “uniform” random variables are shown, where the red line indicates $F_D(\widetilde{D})$.



4.2.2 Normal Distributed Demand

Instead of using the dependency methods described above, a different method can be used for the special case where D is normally distributed. If this is the case, the following method constructs a new correlated random variable:

1. Let $D \sim \mathcal{N}(\mu, \sigma)$ and ρ be the user-defined correlation factor
2. Let $d_1 := \widetilde{D} - \mu$ be the normalization of the previous demand realization $\widetilde{D}$
3. Define $\widehat{D}^* := \rho \cdot d_1 + \sqrt{1-\rho^2} \cdot d_2$, where $d_2 \xleftarrow{R} \mathcal{N}(0, \sigma)$
4. Return the new demand realization $D^* = \widehat{D}^* + \mu$

By the normalized random variable $\mathcal{N}(0, \sigma 0)$, it is possible to construct a new sample $\widehat{D}^*$ with the specific correlation factor ρ, while retaining a mean of 0 and the same variance. The construction of step 4 can be seen as follows:

Given two i.i.d. random variables X_1, X_2 where $\mathbb{E}[X_i] = 0$ for $i = 1, 2$, we wish to construct a third variable Y, such that

$$Y = \alpha \cdot X_1 + \beta \cdot X_2 \text{ such that } \begin{cases} corr(Y, X_1) = \rho \\ \mathbb{E}[Y] = \mathbb{E}[X_1] \\ \mathbb{V}[Y] = \mathbb{V}[X_1] \end{cases} \tag{4.14}$$

By combining the properties above, it follows that

$$\begin{aligned}
\mathbb{E}[Y] &= \alpha \cdot \mathbb{E}[X_1] + \beta \cdot \mathbb{E}[X_2] \\
&= 0 && (4.15) \\
\mathbb{V}[Y] &= \alpha^2 \cdot \mathbb{V}[X_1] + \beta^2 \cdot \mathbb{V}[X_2] \\
\implies 1 &= \alpha^2 + \beta^2 && (4.16) \\
corr(Y, X_1) &= \frac{cov(\alpha \cdot X_1 + \beta \cdot X_2, X_1)}{\sigma(Y) \cdot \sigma(X_1)} \\
&= \frac{\alpha \cdot cov(X_1, X_1) + \beta \cdot cov(X_1, X_2)}{\mathbb{V}[X_1]} \\
&= \alpha && (4.17) \\
\implies \alpha &= \rho \wedge \beta = \pm\sqrt{1 - \rho^2} && (4.18)
\end{aligned}$$

By letting $X_1 = X_2 \leftarrow \widehat{D}$ and $Y \leftarrow \widehat{D}^*$, we obtain our desired result. As such, it becomes clear that the normalization is needed to ensure that $\mathbb{E}[\widehat{D}^*] = 0$. In fact, the reason why this technique only works for normally distributed demand D is because we merely compare the first and second moment when constructing the new dependent random variable. As such, it cannot be ensured that the new random variable follows the same distribution. However, as the sum of independent normally distributed variables is again normal, D^* retains the normal distribution. By comparing for example the moment generating function, this technique could be used with random variables other than the normal distribution. However, this is outside the scope of this thesis.

4.3 Arrival Process

A completely different method for creating demand realizations which was investigated is to use an arrival process. This technique is based on (parts of) simulations used in the field of queueing theory. The main idea is that customers arrive with a particular order, such that the demand on a particular day is given by the sum of all orders of arrived customers. In particular, if the pool of customers is not too large, and the inter-arrival times of the customers are not exponentially distributed, it becomes clear that the demand per day is not independent.

Let $\Gamma = (\gamma_1, \gamma_2, ...)$ be the customer pool, which is the collection of all customers that can possibly arrive. Each customer γ_l with $l = 1, 2, ...$ has the following properties:

Inter-arrival time A_l
: A discrete non-negative random variable indicating the number of days between successive arrivals of customer γ_l. Note that A_l can be constant, which is a special case of a random variable.

Order amount for item i $O_l(i)$
: A random variable indicating the amount of product i customer γ_l orders upon arrival, for $i \in \{1, 2, ..., N\}$. Note that $O_l(i)$ is non-negative, and can be constant if desired. In particular, $O_l(i) \equiv 0$ for every item the customer does not order.

Time since last arrival τ_l
An integer indicating the number of time steps (i.e. days) since the previous arrival. By definition, τ_l is non-negative, and is set to 0 only upon arrival of the customer.

Note that no time indices are required for the inter-arrival time and order amount of each customer; for convenience, it is assumed that these properties are independent through time.[1]

Upon initializing the customer pool, the time since the last arrival t_l is set to a (uniformly chosen) random value between 1 and $\max A_l$. This way, the previous arrival of each customer can be any number of periods ago, so long as it does not exceed the maximum number of periods between arrivals. After initialization, it can be checked which customers arrive on a day:

$$\mathbb{P}(\gamma_l \text{ arrives}) = \mathbb{P}(A_l = t_l \mid A_l > t_l - 1) \qquad \forall l : \gamma_l \in \Gamma \tag{4.19}$$

Upon arrival of customer γ_l, an order Ω_l is placed by that customer, given by

$$\Omega_l := (\Omega_l(1), \Omega_l(2), ..., \Omega_l(n)) \overset{R}{\longleftarrow} (O_l(1), O_l(2), ..., O_l(n)) \qquad \forall l : \gamma_l \in \Gamma \tag{4.20}$$

For ease of notation, let $\Omega_l = \underline{0}$ if customer γ_l does not arrive. Using the above, the full order of item i is given by

$$\widetilde{D}_i = \sum_{\gamma_l \in \Gamma} \Omega_l(i) \qquad \forall i \in \{1, ..., N\} \tag{4.21}$$

As we wish to implement such an arrival process in the scenario tree (Section 3.1), it is required to keep track of a scenario tree of customer pools $\Gamma_{t,j}$: at time t and scenario j, customer pool $\Gamma_{t,j}$ is used to sample C_t demand realizations, where C_t indicates the number of child nodes at time t. The only variable which changes between the different customer pools is the number of days since the last arrival t_l for each of the customers $\gamma_l \in \Gamma_{t,j}$, as seen in the example below.

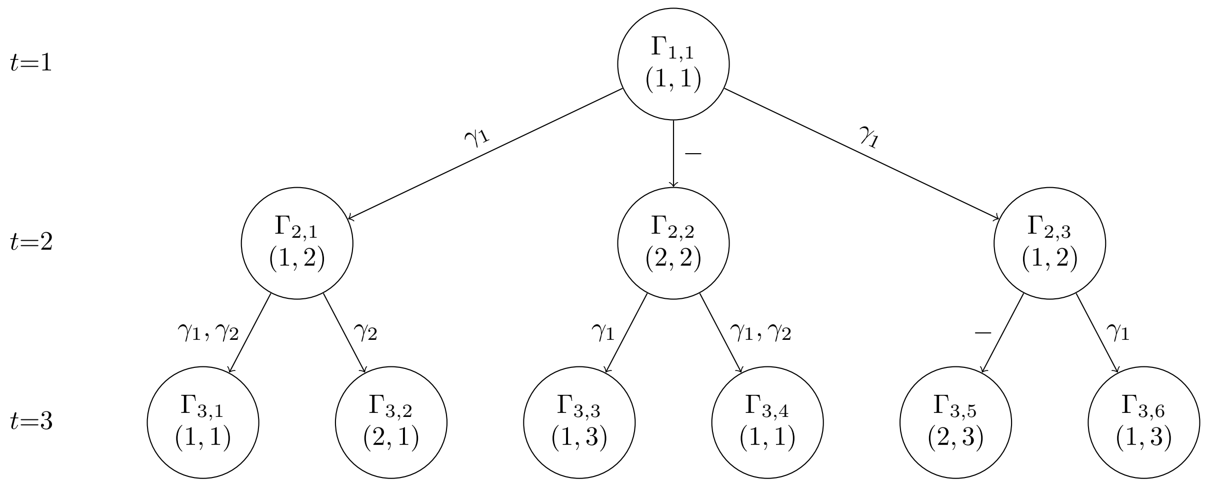


Figure 4.4: Tree of customer pools $\Gamma_{t,j}$ and previous arrivals (t_1, t_2). Each customer pool contains 2 customers γ_1, γ_2 where $A_1 = \mathscr{U}[1,2]$ and $A_2 = \mathscr{U}[2,4]$. Each edge denotes which customers arrive (if any). The order amounts O_k are omitted for readability.

[1]This assumption is not required, and can easily be adjusted in the model if desired.

4.3.1 Parameter Reduction

By the construction of the arrival process, a significant amount of different parameters are allowed: each of the n customers can be unique, with each a particular arrival and order distribution. As the main goal of the arrival process is to enforce a specific correlation, we wish to drastically reduce the number of different parameters, and search for a relation between these parameters and the correlation of successive demands.

Let M denote the number of (identical!) customers, where the inter-arrival distribution of each customer is Uniform$\{1, b\}$ for $b \in \mathbb{N}^+$, and the order amount for each item O_i is constant. Per customer, the average number of days between arrivals is given by $\frac{b+1}{2}$, such that the average demand of item i per day is given by

$$\mathbb{E}[D_i] = M \cdot O_i \cdot \frac{2}{b+1} \tag{4.22}$$

For computing the correlation between successive demands, we investigate a single item ($n = 1$), such that we omit the subscript i. Furthermore, let $O_i = 1$ and $M = 1$.[2] Let D and D' represent the demand of yesterday and today respectively. Both D and D' are defined by (the same) customer with an inter-arrival distribution of Uniform$\{1, b\}$. As such, D is in fact a Bernoulli distribution with a success probability of $p = \frac{2}{b+1}$. This phenomenon is proven in detail in the following section. Observe that

$$\mathbb{E}[D] = \mathbb{E}[D'] = \frac{2}{b+1} \tag{4.23}$$

$$\mathbb{V}[D] = \mathbb{V}[D'] = \frac{2}{b+1} \cdot \left(1 - \frac{2}{b+1}\right) \mathbb{E}[D \cdot D'] \qquad = \mathbb{P}(D = 1 \cap D' = 1)$$

$$= \mathbb{P}(D = 1) \cdot \mathbb{P}(D' = 1 \mid D = 1)$$

$$= \frac{2}{b+1} \cdot \frac{1}{b} \tag{4.24}$$

If the customer arrived yesterday, chances of arriving again today are given by $\frac{1}{b}$. Combining the equations above, it follows that

$$\begin{aligned}
cor(D, D') &= \frac{cov(D, D')}{\sigma(D) \cdot \sigma(D')} \\
&= \frac{\mathbb{E}[D \cdot D'] - \mathbb{E}[D] \cdot \mathbb{E}[D']}{\mathbb{V}[D]} \\
&= \frac{\frac{2}{b(b+1)} - \left(\frac{2}{b+1}\right)^2}{\frac{2 \cdot (b+1-2)}{(b+1)^2}} \\
&= \frac{\frac{2(b+1)}{b(b+1)^2} - \frac{4}{(b+1)^2}}{\frac{2b-2}{(b+1)^2}} \\
&= \frac{\frac{2b+2-4b}{b}}{2b-2} \qquad (4.25) \\
\implies cor(D, D') &= -\frac{1}{b} \vee b = 1 \qquad (4.26)
\end{aligned}$$

The exception $b = 1$ follows from a zero-valued denominator. However, the correlation can easily be computed for $b = 1$, as $D \equiv D' \equiv 1$ every day. While D and D' are in fact equal, they are independent, such that the correlation is equal to 0. For varying values of b, these theoretical results are verified by simulation.

[2]Simulation results show that the correlation remains the same for higher values of M, such that we only prove the result for $M = 1$.

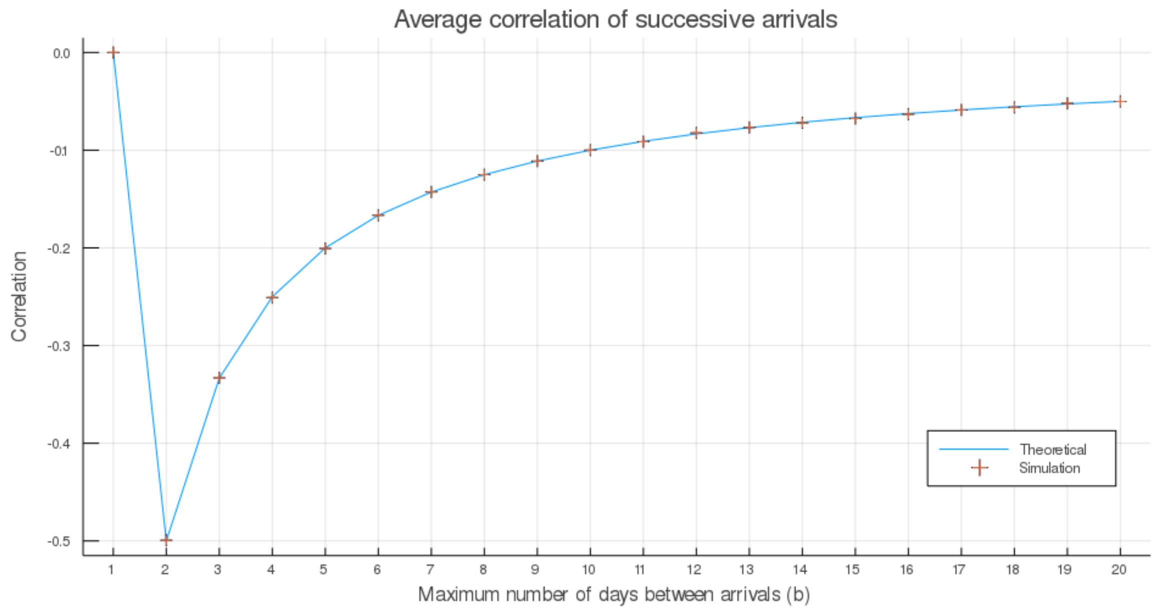


Figure 4.5: Comparison of theoretical and simulated correlation between successive demands, created by an arrival process of a single customer.

It follows that the lowest anti-correlation that can be enforced this way is by setting $b = 2$, resulting in a correlation of $\rho = -0.5$. To obtain even higher anti-correlation, adjusting the inter-arrival distribution to be Uniform$\{2, 3\}$ gives the desired result. Following the same logic as the equations above, it follows that

$$\mathbb{E}[D] = \mathbb{E}[D'] = \frac{2}{5} \tag{4.27}$$

$$\mathbb{V}[D] = \mathbb{V}[D'] = \frac{2}{5} \cdot \frac{3}{5} \tag{4.28}$$

$$\begin{aligned} \mathbb{E}[D \cdot D'] &= \mathbb{P}(D = 1) \cdot \mathbb{P}(D' = 1 \,|\, D = 1) \\ &= 0 \end{aligned} \tag{4.29}$$

$$\begin{aligned} corr(D, D') &= \frac{0 - \left(\frac{2}{5}\right)^2}{\frac{6}{25}} \\ &= -\frac{2}{3} \end{aligned} \tag{4.30}$$

If we wish to create even higher anti-correlation between successive demands, a different approach is required. Examples are changing the arrival distribution, taking a non-deterministic order amount, or adding one or two additional customer types to the process.

Stationary Distribution

For any inter-arrival distribution, the probability of arriving is dependent on the number of days τ since the last arrival. However, in Section 4.3.1, we claim that the inverse of the expected inter-arrival time provides the expected demand per day. To verify this, using the fact that the inter-arrival distribution is Uniform$\{1, b\}$, define a discrete-time Markov chain, where each state indicates the number of days since the last arrival. Then the transition matrix P is as follows:

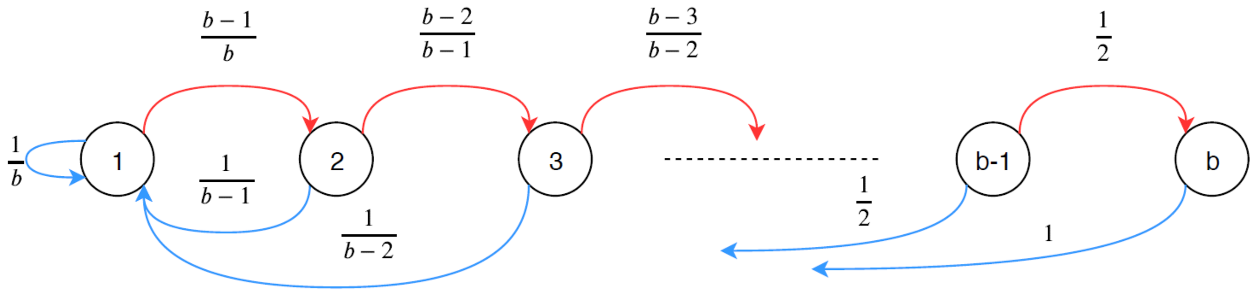


Figure 4.6: A Markov chain showing the behaviour of τ, indicating the number of days since the last arrival for a single customer, where the probability transitions are shown for each connection. All blue arrows point towards the node $\tau = 1$.

$$P = \begin{bmatrix} \frac{1}{b} & \frac{b-1}{b} & 0 & \cdots & 0 \\ \frac{1}{b-1} & 0 & \frac{b-2}{b-1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \frac{1}{2} & 0 & 0 & \ddots & \frac{1}{2} \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{4.31}$$

Observe that each state τ has two outgoing connections: one representing an arrival (the blue arrow connecting to 1) and the other representing no arrival, such that τ is increased by 1. Using the above, the steady-state distribution can be found using the global balance equations:

$$\begin{aligned} \pi_b &= \frac{1}{2} \cdot \pi_{b-1} \\ &= \frac{1}{2} \cdot \frac{2}{3} \cdot \pi_{b-2} \\ &= \frac{1}{3} \cdot \pi_{b-2} && (4.32) \\ \implies \pi_b &= \frac{1}{k+1} \cdot \pi_{b-k} \qquad \forall k \in \{0, 1, ..., b-2\} && (4.33) \end{aligned}$$

All nodes have a non-negative transition probability to node $\tau = 1$, such that

$$\begin{aligned} \pi_1 &= \frac{1}{b} \cdot \pi_1 + \sum_{\tau=2}^{b} \frac{1}{b-\tau+1} \cdot \pi_\tau \\ &= \frac{\pi_1}{b} + \sum_{\tau=2}^{b} \frac{1}{b-\tau+1} \cdot (b-\tau+1) \cdot \pi_b \\ \frac{b-1}{b} \cdot \pi_1 &= (b-1) \cdot \pi_b \\ \implies \pi_1 &= b \cdot \pi_b && (4.34) \end{aligned}$$

Lastly, the normalization formula is applied, resulting in

$$\begin{aligned}
1 &= \sum_{\tau=1}^{b} \pi_\tau \\
&= \pi_b \cdot \left(b + \sum_{\tau=2}^{b} (b - \tau + 1) \right) \\
&= \pi_b \cdot \left(b + \frac{b \cdot (b-1)}{2} \right) && (4.35) \\
\implies \pi_\tau &= \frac{2b - 2\tau + 2}{b \cdot (b+1)} \qquad \forall \tau \in \{1, 2, ..., b\} && (4.36)
\end{aligned}$$

Using the stationary distribution and the probability of arriving for each τ, it follows that the the probability of customer C arriving on a particular day is given by

$$\begin{aligned}
\mathbb{P}(C \text{ arrives}) &= \sum_{t=1}^{b} \mathbb{P}(C \text{ arrives} \,|\, \tau = t) \cdot \pi_t \\
&= \sum_{t=1}^{b} \frac{1}{b - \tau + 1} \cdot \frac{2(b - \tau + 1)}{b \cdot (b+1)} \\
&= \sum_{t=1}^{b} \frac{2}{b(b+1)} \\
&= \frac{2}{b+1} && (4.37)
\end{aligned}$$

Stationary Distribution - Revised

An alternative, more hands-on method of determining the stationary distribution is as follows: assume that, upon arrival of a customer, the next arrival is immediately planned. These next arrivals are uniformly distributed between 1 and b. Observe that all possible numbers of days τ since the last arrival can be denoted as follows:

$$\text{Next arrival:} \begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ b \end{bmatrix} \implies \tau \in \begin{pmatrix} 1 & & & & \\ 1 & 2 & & & \\ 1 & 2 & 3 & & \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & 2 & 3 & \cdots & b \end{pmatrix} \tag{4.38}$$

In the matrix notation, observe that each number τ occurs $b - \tau + 1$ times. As each next arrival is chosen uniformly between 1 and b, each of the $\frac{b \cdot (b+1)}{2}$ numbers in the matrix has equal probability of appearing. In terms of the stationary distribution,

$$\pi_\tau = (b - \tau + 1) \left(\frac{b \cdot (b+1)}{2} \right)^{-1} \qquad \forall \tau \in \{1, 2, ..., b\} \tag{4.39}$$

The equation above is in fact identical to Equation (4.36), which verifies our previous conclusions.

5 Implementation

The program described in this report has been written in Julia (v.1.2.0).[Bezanson et al., 2017] The code is written in an object-oriented manner, as indicated in Section 5.1. Moreover, we use the property of *multiple dispatch* for several different methods, as is common in Julia code.

By creating different objects for different tasks, we are able to simulate an environment in which a fair comparison can be made between different decision methods. A particular Decider does not have information regarding the upcoming uncertainty, and uses a DemandModel to create a DemandPrediction. Afterwards, a separate Simulator is used to create a (series of) DemandRealizations, which are the same for all compared Deciders.

For the stochastic programming approach, in order to build and solve mixed-integer programs, Julia for Mathematical Programming (JuMP) is used.[Dunning et al., 2017] The reason for using JuMP is because it provides a clear way of constructing linear programs, where we can easily add or modify existing variables and constraints. Furthermore, it supports several different solvers, such as Gurobi and CPLEX. In our case, Gurobi was chosen as the solver, primarily because of its fast solving speed.[Gurobi Optimization, 2019]

5.1 Structures

Several different objects have been used to provide a clear structure for the program. These objects are defined as *structs* (similar to *classes* in Python), where each struct has several fields containing its properties. Next to that, most structs have a number of functions associated with it[1]. Figures 5.2, 5.3, and 5.4 provide an overview of the structs and functions used. The structures and corresponding functions are categorized into three main categories, explained in detail in the following sections.

Throughout this section, we use diagrams corresponding to the Unified Modeling Language (UML). An example of such a diagram is provided in Figure 5.1. Most structs used are subtypes of a (user-defined) abstract type. This subtype hierarchy is used primarily for *multiple dispatch*, as explained in Section 5.2. Each block in the diagram contains the fields and functions belonging to a particular struct. A distinction is made between public and private fields (or functions) - this is mainly to indicate which properties are not directly accessed by other structs.

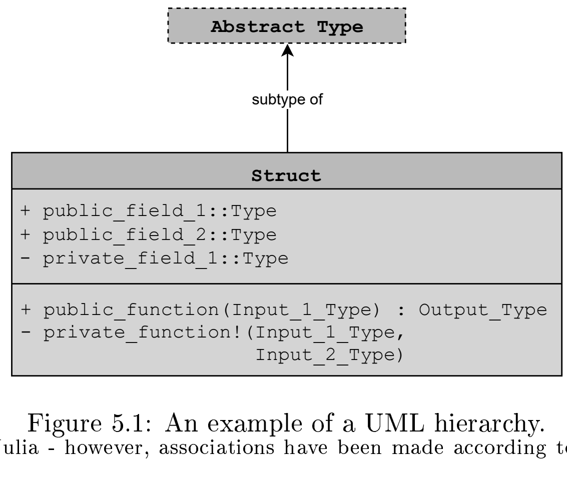


Figure 5.1: An example of a UML hierarchy.

[1]Functions are not necessarily associated with a specific struct in Julia - however, associations have been made according to the in- or output of a function.

5.1.1 Basic Objects

The objects used throughout the simulation are described first. These objects are not hierarchical, but are often used by the other objects.

ScenarioTree Contains information on the size of the scenario tree. Primarily used by the StochasticProgram.

Warehouse Contains most model parameters, such as penalty times (costs) and maximum inventory levels.

State Represents the current state (of the Warehouse), and is used to keep track of (past) demands.

Results Stores important information resulting from the simulation, such as computation time and costs.

Simulator Contains information on the simulated reality, such as the (true) DemandModel and the horizon to be simulated. Note that this DemandModel is not necessarily the same as what the Deciders use.

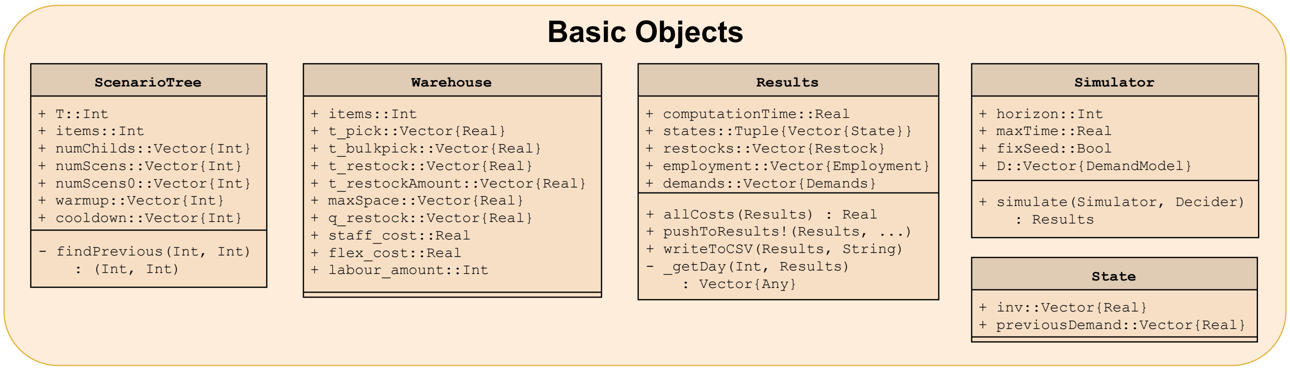

Basic Objects
ScenarioTree
+ T::Int
+ items::Int
+ numChilds::Vector{Int}
+ numScens::Vector{Int}
+ numScens0::Vector{Int}
+ warmup::Vector{Int}
+ cooldown::Vector{Int}
- findPrevious(Int, Int) : (Int, Int)
Warehouse
+ items::Int
+ t_pick::Vector{Real}
+ t_bulkpick::Vector{Real}
+ t_restock::Vector{Real}
+ t_restockAmount::Vector{Real}
+ maxSpace::Vector{Real}
+ q_restock::Vector{Real}
+ staff_cost::Real
+ flex_cost::Real
+ labour_amount::Int
Results
+ computationTime::Real
+ states::Tuple{Vector{State}}
+ restocks::Vector{Restock}
+ employment::Vector{Employment}
+ demands::Vector{Demands}
+ allCosts(Results) : Real
+ pushToResults!(Results, ...)
+ writeToCSV(Results, String)
- _getDay(Int, Results) : Vector{Any}
Simulator
+ horizon::Int
+ maxTime::Real
+ fixSeed::Bool
+ D::Vector{DemandModel}
+ simulate(Simulator, Decider) : Results
State
+ inv::Vector{Real}
+ previousDemand::Vector{Real}

Figure 5.2: An overview of the basic objects.

5.1.2 Deciders

When simulating the inventory model, a Decision needs to be made on a daily basis. The Decider object is used to provide a Decision (in our current situation, a Restock and / or an Employment), where different methods are used depending on the type. Furthermore, the Costs (as a consequence of each Decision) are split into employment, restock, direct- and bulk-pick costs, and subsequently stored in a separate struct.

Decider Abstract type. Used to provide a Decision.

> **StochasticProgram** Builds and solves a JuMP model to find the optimal Decision.
>
> **Policy** Requires one-time initialization of policy values, found using its PolicyApproach.

PolicyApproach Abstract type. Used to provide Policy values.

> **PolicyHeuristic** A heuristic approach, which determines the optimal restock inventory in the single-stage problem.
>
> **PolicySP** Uses a similar approach as StochasticProgram, but instead optimizes policy values.

Decision Abstract type. Provided as a result by the Decider.

> **Restock** The amount of containers to restock from the bulk area to the picking area.

Employment The number of staff- and flex workers to hire on a particular day.

Costs Abstract type. Distinguished in employment, restock, direct- and bulk-pick costs, primarily used for analysis and debugging.

CostsItem Costs of a single item in the Warehouse, used by PolicyHeuristic. Employment costs are not incorporated as these cannot be computed for a single item.

CostsVector Costs for all items in the Warehouse, used by the Simulator.

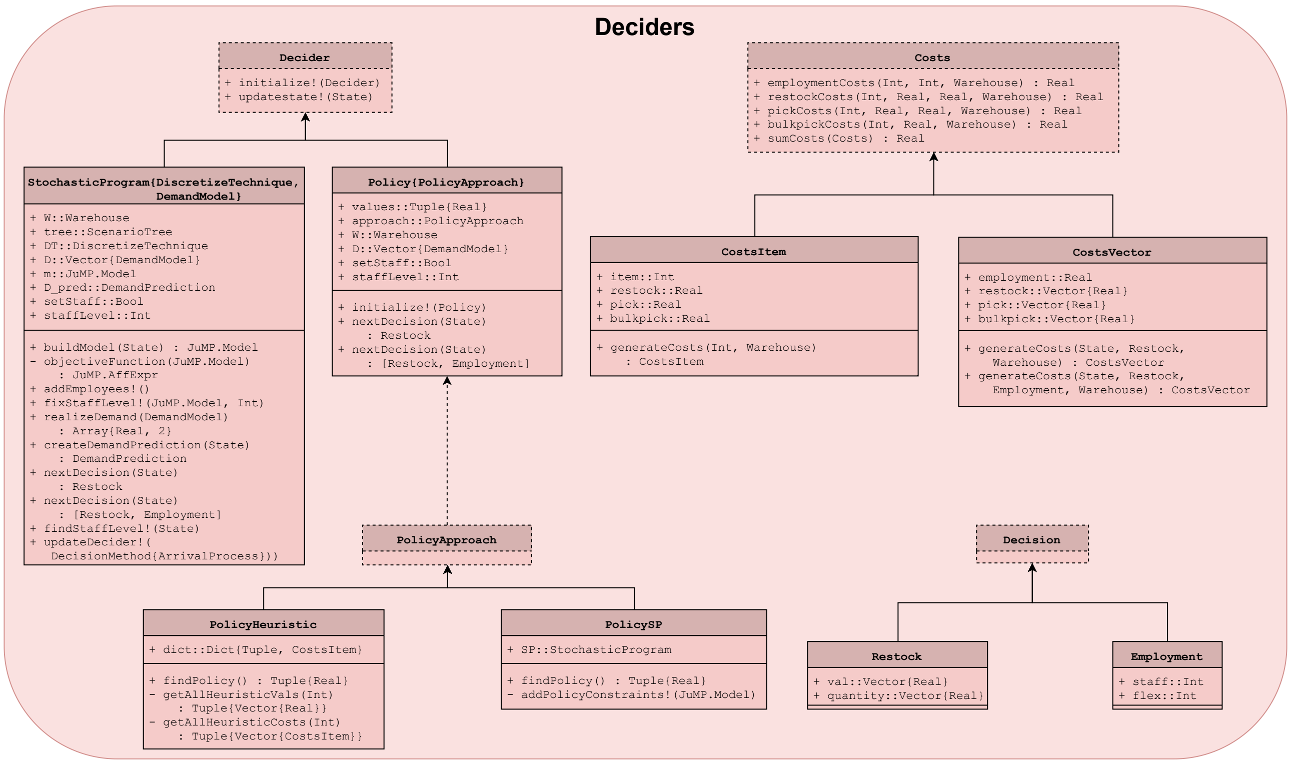

Deciders
Decider
+ initialize!(Decider)
+ updatestate!(State)
Costs
+ employmentCosts(Int, Int, Warehouse) : Real
+ restockCosts(Int, Real, Real, Warehouse) : Real
+ pickCosts(Int, Real, Real, Warehouse) : Real
+ bulkpickCosts(Int, Real, Warehouse) : Real
+ sumCosts(Costs) : Real
StochasticProgram{DiscretizeTechnique, DemandModel}
+ W::Warehouse
+ tree::ScenarioTree
+ DT::DiscretizeTechnique
+ D::Vector{DemandModel}
+ m::JuMP.Model
+ D_pred::DemandPrediction
+ setStaff::Bool
+ staffLevel::Int
+ buildModel(State) : JuMP.Model
- objectiveFunction(JuMP.Model) : JuMP.AffExpr
+ addEmployees!()
+ fixStaffLevel!(JuMP.Model, Int)
+ realizeDemand(DemandModel) : Array{Real, 2}
+ createDemandPrediction(State) : DemandPrediction
+ nextDecision(State) : Restock
+ nextDecision(State) : [Restock, Employment]
+ findStaffLevel!(State)
+ updateDecider!(DecisionMethod{ArrivalProcess})
Policy{PolicyApproach}
+ values::Tuple{Real}
+ approach::PolicyApproach
+ W::Warehouse
+ D::Vector{DemandModel}
+ setStaff::Bool
+ staffLevel::Int
+ initialize!(Policy)
+ nextDecision(State) : Restock
+ nextDecision(State) : [Restock, Employment]
CostsItem
+ item::Int
+ restock::Real
+ pick::Real
+ bulkpick::Real
+ generateCosts(Int, Warehouse) : CostsItem
CostsVector
+ employment::Real
+ restock::Vector{Real}
+ pick::Vector{Real}
+ bulkpick::Vector{Real}
+ generateCosts(State, Restock, Warehouse) : CostsVector
+ generateCosts(State, Restock, Employment, Warehouse) : CostsVector
PolicyApproach
Decision
PolicyHeuristic
+ dict::Dict{Tuple, CostsItem}
+ findPolicy() : Tuple{Real}
- getAllHeuristicVals(Int) : Tuple{Vector{Real}}
- getAllHeuristicCosts(Int) : Tuple{Vector{CostsItem}}
PolicySP
+ SP::StochasticProgram
+ findPolicy() : Tuple{Real}
- addPolicyConstraints!(JuMP.Model)
Restock
+ val::Vector{Real}
+ quantity::Vector{Real}
Employment
+ staff::Int
+ flex::Int

Figure 5.3: The hierarchy of the Deciders section.

5.1.3 Demands

Throughout the simulation, demand is simulated to indicate orders of customers, such that the number of products in stock is reduced. The most important functionalities are to create DemandRealizations (called by the Simulator) and DemandPredictions (called by the Decider). The exact way how these are constructed depends on the (type of) DemandModel used.

DemandModel Abstract type. Used to create DemandRealizations and DemandPredictions.

- **Demand_IID** Creates independent and identically distributed samples.
- **Demand_NonIID** Uses a DependencyEnforcer to form dependent samples.
- **ArrivalProcess** Simulates a process of arriving customers, each having a particular order of items.

DependencyEnforcer Abstract type. Enforces correlation between samples; used only in the non-i.i.d. DemandModel.

- **EqualityProbability** Creates a new sample which is equal to the previous sample with a certain probability.
- **UniformNoise** Creates a new sample by adding uniform 'noise' to the (normalized) previous sample.
- **NormalCorrelation** Correlates normally-distributed 'random' samples, retaining the properties of the normal distribution.

Customer Used only in the ArrivalProcess; has properties to determine inter-arrival time and order amounts.

DemandRealization Contains vector of demand values, indicating the (simulated) reality.

DemandPrediction Used in Stochastic Programming techniques to create a ScenarioTree of demands.

DiscretizeTechnique Abstract type. Indicates method used to create DemandPrediction.

- **RandomSample** Randomly sample values based on the provided DemandModel.
- **Percentiles** Use percentiles of the DemandModel distribution.

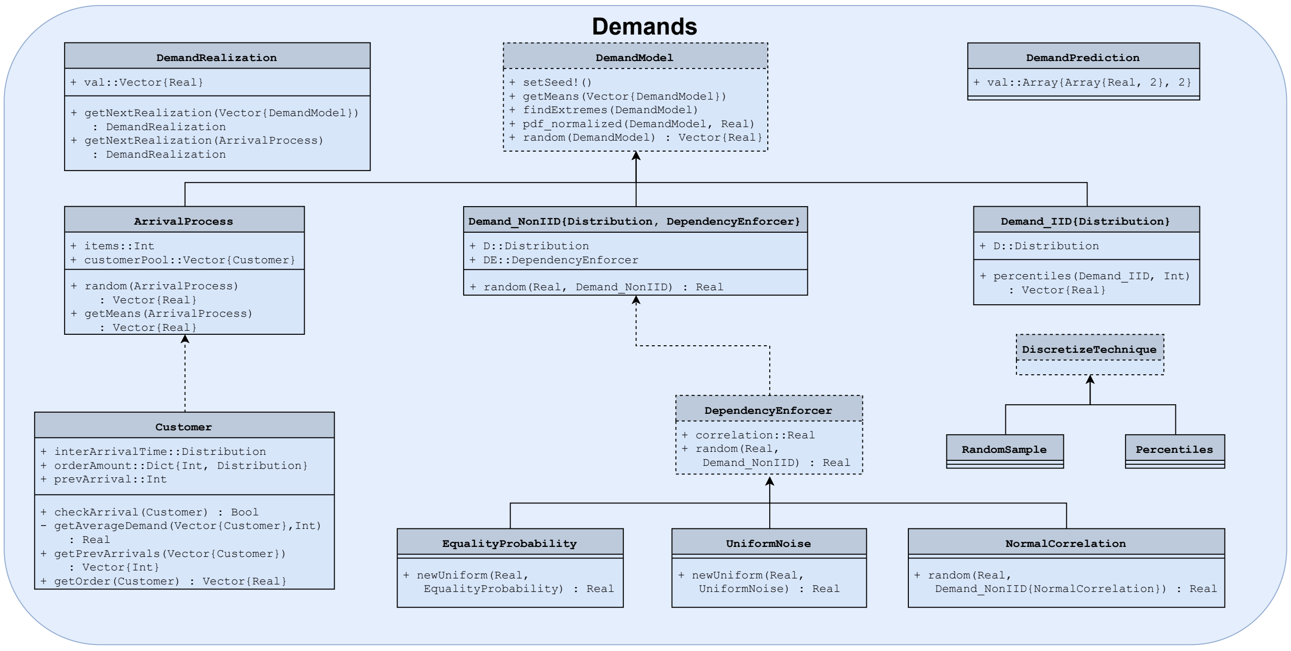

Demands
DemandRealization
+ val::Vector{Real}
+ getNextRealization(Vector{DemandModel}) : DemandRealization
+ getNextRealization(ArrivalProcess) : DemandRealization
DemandModel
+ setSeed!()
+ getMeans(Vector{DemandModel})
+ findExtremes(DemandModel)
+ pdf_normalized(DemandModel, Real)
+ random(DemandModel) : Vector{Real}
DemandPrediction
+ val::Array{Array{Real, 2}, 2}
ArrivalProcess
+ items::Int
+ customerPool::Vector{Customer}
+ random(ArrivalProcess) : Vector{Real}
+ getMeans(ArrivalProcess) : Vector{Real}
Demand_NonIID{Distribution, DependencyEnforcer}
+ D::Distribution
+ DE::DependencyEnforcer
+ random(Real, Demand_NonIID) : Real
Demand_IID{Distribution}
+ D::Distribution
+ percentiles(Demand_IID, Int) : Vector{Real}
DiscretizeTechnique
RandomSample
Percentiles
DependencyEnforcer
+ correlation::Real
+ random(Real, Demand_NonIID) : Real
Customer
+ interArrivalTime::Distribution
+ orderAmount::Dict{Int, Distribution}
+ prevArrival::Int
+ checkArrival(Customer) : Bool
- getAverageDemand(Vector{Customer},Int) : Real
+ getPrevArrivals(Vector{Customer}) : Vector{Int}
+ getOrder(Customer) : Vector{Real}
EqualityProbability
+ newUniform(Real, EqualityProbability) : Real
UniformNoise
+ newUniform(Real, UniformNoise) : Real
NormalCorrelation
+ random(Real, Demand_NonIID{NormalCorrelation}) : Real

Figure 5.4: The hierarchy of the Demands section.

5.2 Functions

The main function of our model is a simulation using a receding time horizon (see Section 2.2.1). All objects required are initialized (the Decider in particular), and afterwards the main loop is initiated. At each step, the Decider provides the 'optimal' decision, after which the actual demand is realized, the current state is updated, and the loop is repeated. The loop is terminated once the complete horizon has been simulated, or the computation time exceeds its threshold. A high-level overview of this function is provided in Figure 5.5.

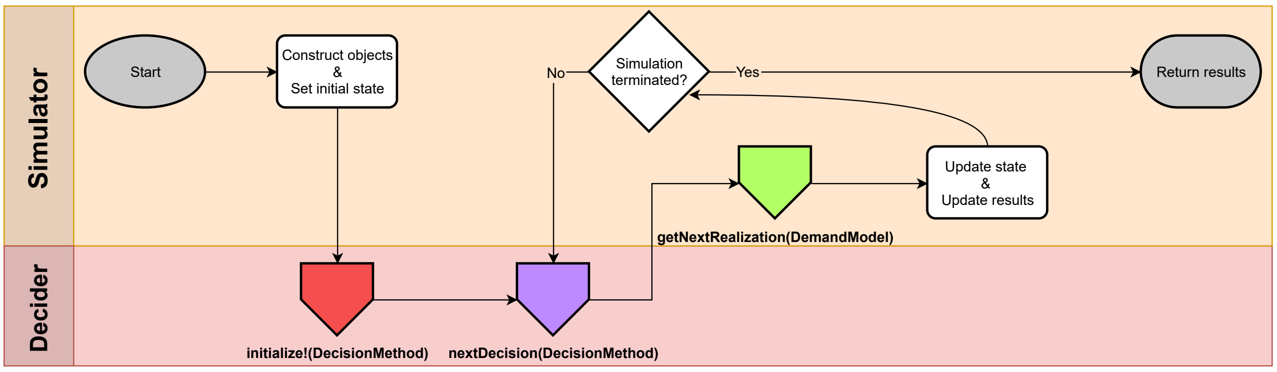

Simulator
Decider
Start
Construct objects & Set initial state
No
Simulation terminated?
Yes
Return results
Update state & Update results
getNextRealization(DemandModel)
initialize!(DecisionMethod)
nextDecision(DecisionMethod)

Figure 5.5: The simulation function, as initiated by the Simulator. See Figures 5.6, 5.7 and 5.8 for the emphasized functions.

The highlighted functions in the figure above are illustrated in the remainder of this section. All of these functions make use of the *multiple dispatch* property: depending on the type of its input, a different method is initiated. Each function shows what abstract type is required (in our example: either a DecisionMethod or DemandModel), where each subtype results in a different method being called.[2]

[2]These methods do not necessarily need to be defined on subtypes, but it gives a clearer overview of the hierarchical structure of the code.

The first is the *intialize!* function, which methods are provided in Figure 5.6. The exclamation mark is added to indicate that the input is modified in-place by the function. Each of the separate methods returns a result of a different type: the StochasticProgram does not require any (prior) initialization, hence returning nothing. However, both Policy Deciders require policy values, which are returned by their corresponding initialize! methods. These policy values are added directly to the policy decider struct.

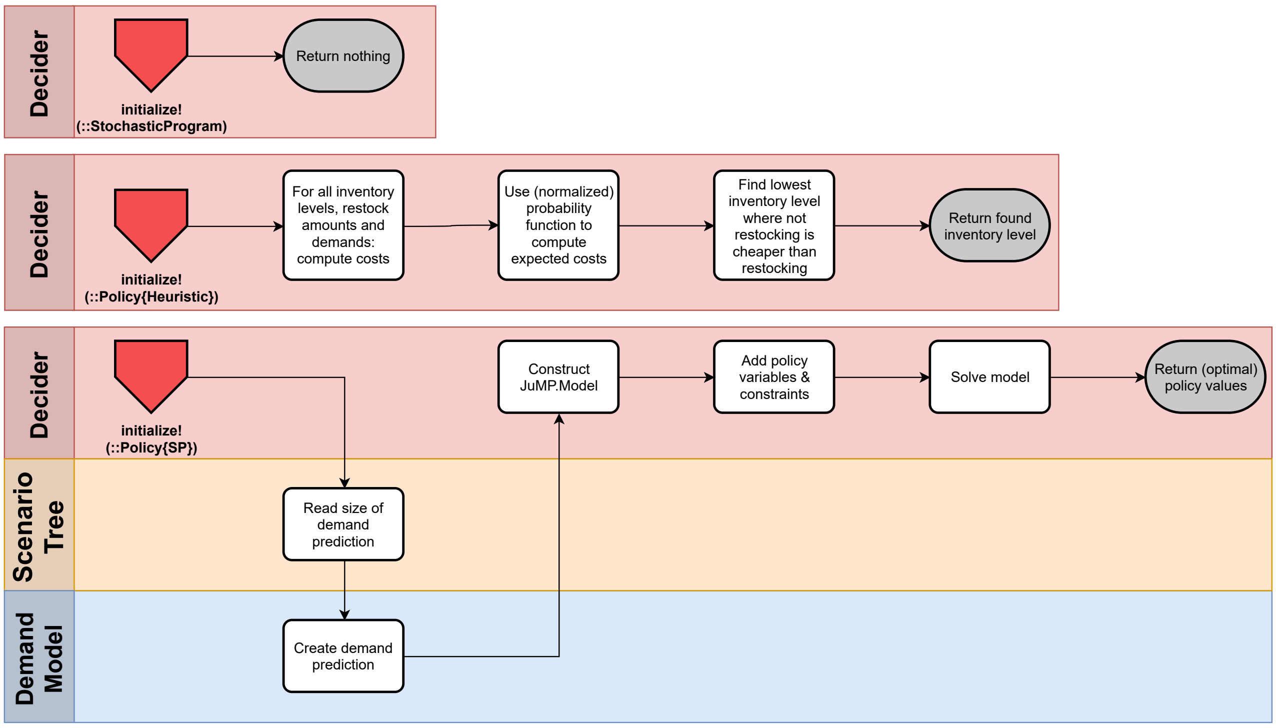

Decider
initialize! (::StochasticProgram)
Return nothing
Decider
initialize! (::Policy{Heuristic})
For all inventory levels, restock amounts and demands: compute costs
Use (normalized) probability function to compute expected costs
Find lowest inventory level where not restocking is cheaper than restocking
Return found inventory level
Decider
initialize! (::Policy{SP})
Construct JuMP.Model
Add policy variables & constraints
Solve model
Return (optimal) policy values
Scenario Tree
Read size of demand prediction
Demand Model
Create demand prediction

Figure 5.6: The initialize! methods, which perform significantly different depending on whether the Decider is StochasticProgram or Policy based.

The second function is the *nextDecision* function, which is called by the Decider in every iterative step of the simulation loop. A Policy Decider merely requires a simple comparison between the inventory level and its (precomputed) policy value. The StochasticProgram, however, builds a scenario tree, creates a demand prediction, and subsequently constructs and solves a MILP in order to provide a restock decision. See Figure 5.7 for an overview of these methods. Note that creating a new DemandPrediction is quite similar to generating a new DemandRealization, as described below.

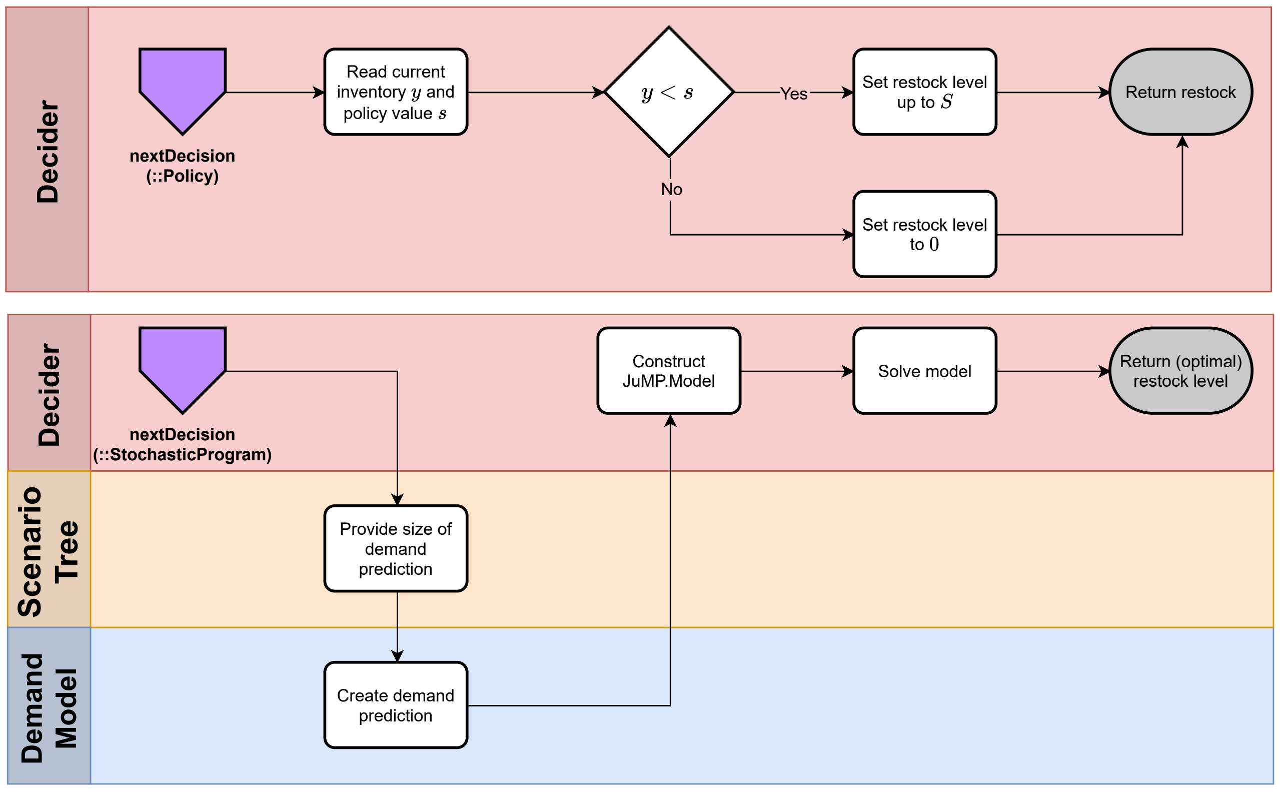

Decider
nextDecision (::Policy)
Read current inventory y and policy value s
$y < s$
Yes
No
Set restock level up to S
Set restock level to 0
Return restock
Decider
Scenario Tree
Demand Model
nextDecision (::StochasticProgram)
Provide size of demand prediction
Create demand prediction
Construct JuMP.Model
Solve model
Return (optimal) restock level

Figure 5.7: The nextDecision methods, corresponding to the two different Decider types.

The third function is the *getNextRealization* function, which is also called in every iterative step of the simulation loop. However, this function is independent of the specific Decider, as it is called by the Simulator. As described in Chapter 4, there are several possible methods for sampling demand. For the Demand_NonIID function, one step is to use the DependencyEnforcer to generate a new sample: details of how these are used can be found in Section 4.2.

Lastly, while not directly mentioned in the previous function, we can add employees to the model. The two main changes made to the functions above are as follows:

initialize! For any Decider, the initialize! function must also return the (optimal) staffing level; in particular, there is in fact an initialization required for the Stochastic Program, as a nextDecision function is called to return a staff level. In order to add such staffing properties, the addEmployees! function is used prior to the model optimization.

nextDecision Any Decider must also return the number of flex workers as the next Decision (together with the next Restock). For the Stochastic Program, the addEmployees! function is used. For any Policy Decider, flex workers are hired to ensure that all items are still restocked (if the staff cannot restock all items themselves).

The addEmployee! function as mentioned above is displayed in Figure 5.9. Note that, besides adding these variables and constraints, the staff level is fixed (as it is optimized during initialization).

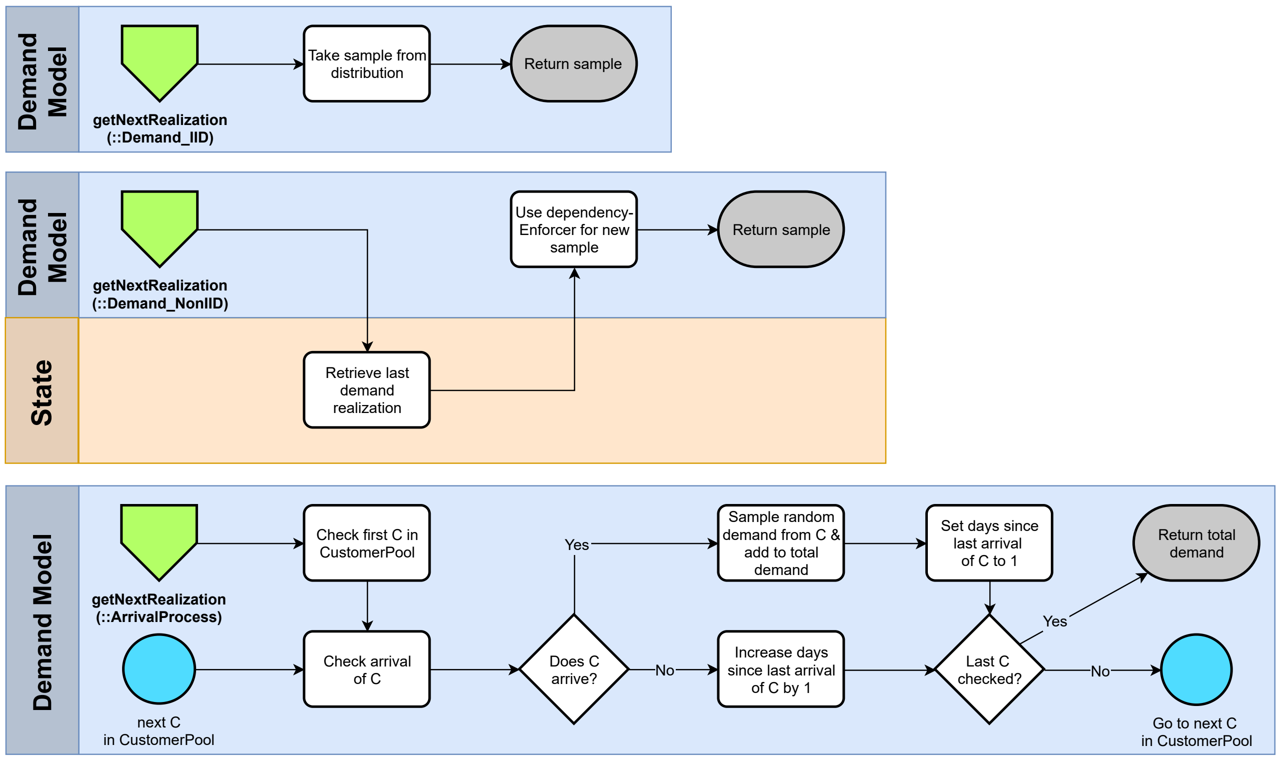

Demand Model
getNextRealization (::Demand_IID)
Take sample from distribution
Return sample
Demand Model
getNextRealization (::Demand_NonIID)
Use dependency-Enforcer for new sample
Return sample
State
Retrieve last demand realization
Demand Model
getNextRealization (::ArrivalProcess)
Check first C in CustomerPool
Yes
Sample random demand from C & add to total demand
Set days since last arrival of C to 1
Return total demand
next C in CustomerPool
Check arrival of C
Does C arrive?
No
Increase days since last arrival of C by 1
Last C checked?
Yes
No
Go to next C in CustomerPool

Figure 5.8: The getNextRealization methods, corresponding to the different DemandModel types used. Note that the ArrivalProcess describes the demand for several items, while the other DemandModels describe the demand of a single item.

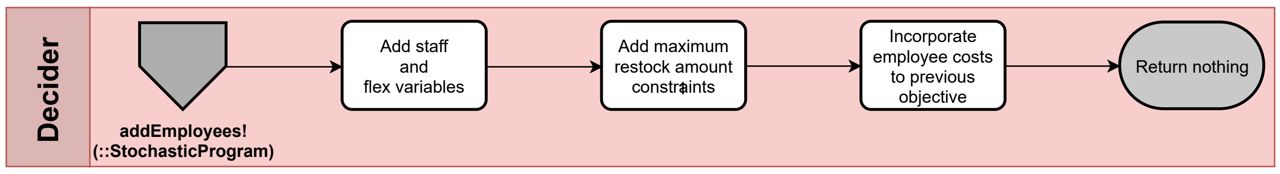

Decider
addEmployees! (::StochasticProgram)
Add staff and flex variables
Add maximum restock amount constraints
Incorporate employee costs to previous objective
Return nothing

Figure 5.9: The addEmployees! function, used in the initialize! and nextDecision functions of the Stochastic Program, but only if employees are modeled.

Besides the functions described above, there are many more used in the code (even more if we count all methods per function). For the full list of functions and their documentation, see the code itself. A shortlist of important functions is as follows:

createDemandPrediction
Effectively creates a large number of demand realizations, but stores it in such a way that it represents a scenario tree of (predicted) demands. For any demand that is not i.i.d., the dependency between successive demand realizations is taken into account.

buildModel
Used only with StochasticProgram solvers: constructs the entire MILP in JuMP, and accesses the (private) function buildObjective for the objective value in particular.

findPrevious
Used only in the building of the linear program by the ScenarioTree: it determines the previous scenario and child node index j^*, k^* by using Equations (3.3) and (3.4).

generateCosts
Combines the separate functions of costs types ((employment), restock, direct- and bulk-pick), and constructs a CostsVector object.

updateDecider!
Does not adjust anything in most cases, but is required if both the Decider and the Simulator use an ArrivalProcess to simulate demand. If this occurs, the Decider is updated such that it has updated information on the previous arrivals of all Customers.

updateState!
Updates the State (the current inventory level and the newest DemandRealization) using the provided Decision and DemandRealization.

pushToResults!
Updates the Results object by adding the (newest) DemandRealization, State, Decision, and computation time.

6 | Results

Having described all aspects of the model and simulation, we now investigate and discuss various results in more detail. These results are all generated using simulations. This way, we are able to emphasize specific behaviour of deciders by choosing certain parameter values. First, we investigate the behaviour of different deciders on a high level. These are among the first results generated from this project, such that these in turn were used in the remainder of the design process. Next, we demonstrate the power of the stochastic program in two separate examples, constructed in such a way that a standard (heuristic) policy has trouble making optimal choices. Lastly, we investigate a large-scale garbage collection problem. Here, we wish to determine how the stochastic program is able to make decisions, in particular when limiting the total amount of computation time allowed.

6.1 Initial Comparison Deciders

As a first result using the simulation, an initial comparison between deciders has been made. In addition to the deciders discussed in Chapter 2, we add an all-knowing decider. This all-knowing decider can be seen as a stochastic program without nonanticipativity constraints: it can make decisions based on future uncertainty, such that it always makes the best possible decision. While such a decider is not realistic, it gives an idea of how far our decider solutions are from the true optimum. Next to that, it gives a bound on how much we should (theoretically) be willing to pay in order to be able to predict the future with complete certainty.

The initial results were constructed with several parameters not initiated, as these were not added to the model at the time of these simulations. The parameters which were not initiated are the staffing variables σ and f, as well as the restocking time per item A_i. Next to that, a simple i.i.d. demand distribution was used for a single item. For the simulator, the most important parameter settings are

$$\tau = 14 \qquad\qquad D \sim \text{Gamma}(17, 2)$$

For the scenario tree, the settings used are

$$N = 1 \qquad\qquad C_t = [5, 5, 3, 2, 1, 1, 1, 1]$$

As the chosen horizon τ is quite short, we use a Monte Carlo simulation of multiple runs, where each run has different demand realizations and a randomly chosen initial inventory level. For each run, these demands and initial inventories are used for all compared deciders, such that a fair comparison is made. These results are plotted in Figure 6.1, and an overview of some of the basic results are provided in Table 6.1. For the policy deciders used, either a stochastic program (SP) is used to determine the policy values, or they are determined using a heuristic algorithm (Section 2.4.1).

Table 6.1: Overview of simulation results from several deciders. For the costs, the mean and standard deviation are given.

Decider	**Costs**		**Time** (s)
All-knowing	613	(± 61)	< 0.1
Stochastic program	654	(± 113)	89
Policy - SP	656	(± 167)	17
Policy - Heuristic	693	(± 163)	0.1

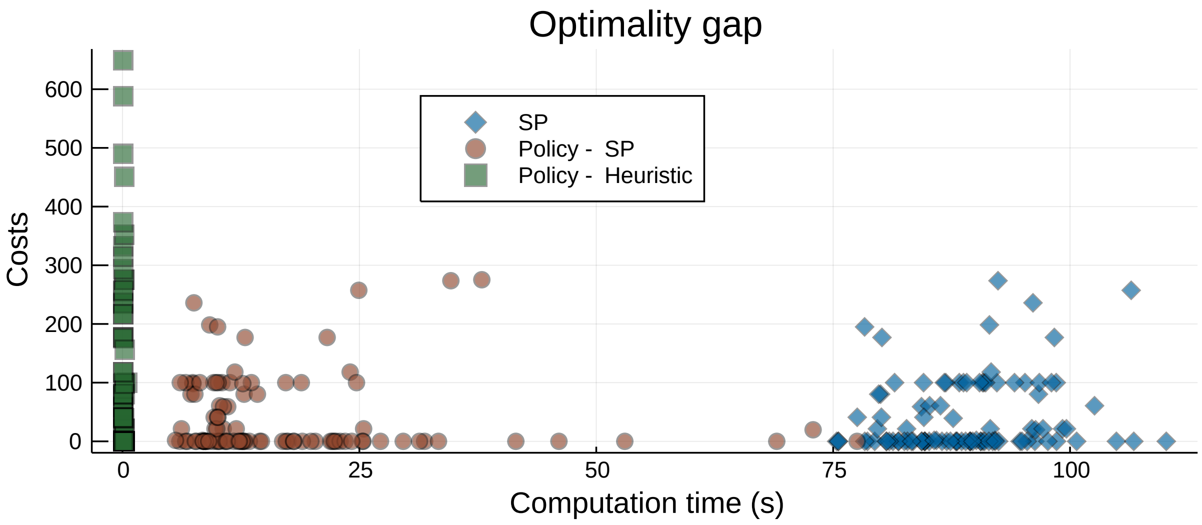


Figure 6.1: The difference in costs between several deciders and the all-knowing decider.

On of the first things which becomes clear is that the stochastic program is only marginally better than the policy decider with SP-determined policy values. Even though the stochastic program could perform much better than any heuristic approach, note that a base-stock policy is almost the best thing we can do for i.i.d. demands. If the optimal restock threshold is found, there is no reason to deviate from this policy on subsequent time steps, as the (distribution of) demand at each time step is independent from all other demands. However, as the scenarios of the scenario tree are randomly generated, the SP is able to make different decisions for the same inventory level. For example, at an inventory level of 30, the SP decides to restock about 50% of the time. This behaviour is demonstrated in Figure 6.2.

Because of this ability to make probabilistic decisions, the stochastic program can attain slightly lower mean costs, and a significantly lower standard deviation than the policy deciders. However, with an average computation time of more than 5 times as long, one could argue that the marginal decrease in costs is not worth the required computations. The same could be said about the policy deciders: on the one hand, the SP is able to find policy values which result in a decrease in costs of about 6% in comparison to the heuristic policy decider. On the other hand, the computation time required for only 1 item in the warehouse is already about 17 seconds, let alone if we wish to optimize the policy values for several hundred items.

As described the policy deciders have less decision freedom than the stochastic program, they can never attain the same optimal decisions. To compare: if the upcoming demand would be known beforehand for the policy decider and the stochastic program, then the average costs resulting from the stochastic program are about 4% less than what the optimal policy value provides. By indicating what the (expected) costs are for several different policy values, we can indicate how well the policy deciders perform. These results are plotted in Figure 6.3.

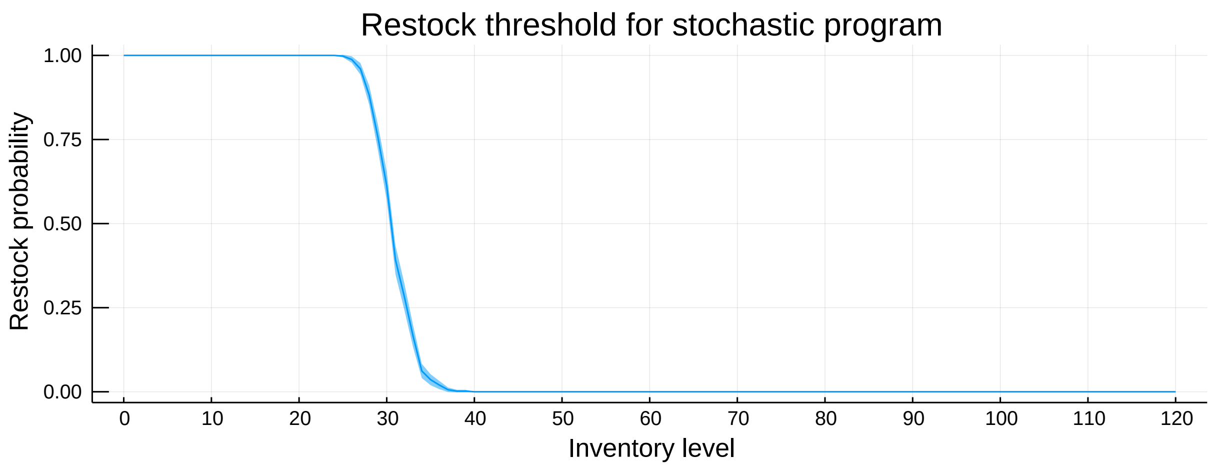


Figure 6.2: The probability of restocking for different inventory levels, as found by the stochastic program. A tree of depth $T = 1$ with $C_1 = 20$ scenarios is used, where the deviation is computed over 500 runs.

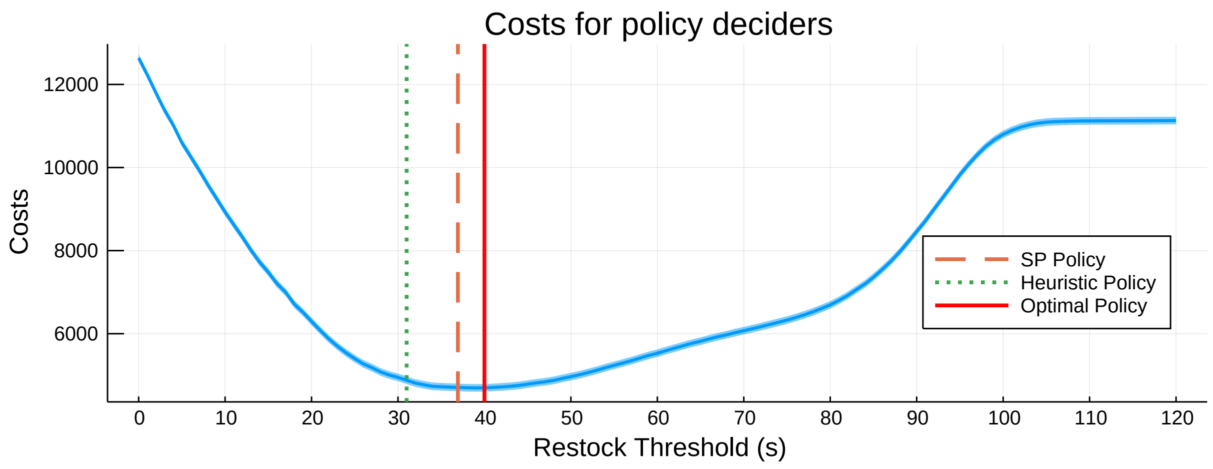


Figure 6.3: The expected costs for varying restock threshold values, and an indication of the policy values found by different deciders. The simulation horizon τ is increased to 100, in order to reduce the effect of the randomized initial inventory.

From these initial results, we concluded that the problem should be increased in difficulty for stochastic programming to be worth the computational costs. As the base stock policies are (near)-optimal for i.i.d. demand, we added several methods to create a correlation between successive demands (Sections 4.2 and 4.3). Next to that, we added complexity to the restocking costs (Section 3.5.2), and added employee costs and constraints. By doing this, the computational cost of solving a stochastic program is increased, but the idea is that the heuristic approaches are not able to cope with all of these additional complexities.

In order to test these new additions to the model, two separate examples have been investigated, each made for a particular aspect of the decider. These examples are described in the sections below.

6.2 Example: Predicting Customer Arrivals

An example which we investigate in detail is constructed to illustrate the power of stochastic programming compared to a simpler base stock policy. For this end, only a single item is considered ($i = 1$). For the demand model, an arrival process is used, where the customer pool Γ contains two identical customers γ_1, γ_2, each having a constant order amount of 1 upon arrival, and a discrete-uniform inter-arrival time $A_l \sim \mathscr{U}[2, 3]$. Using this, the average daily demand that follows is

$$\mathbb{E}[D_t] = 2 \cdot \left(\frac{5}{2}\right)^{-1} = \frac{4}{5} \quad \forall t \in \{1, 2, ..., T\} \tag{6.1}$$

As the demand on a particular day is at most 2, the maximum inventory space is set to 2 as well. On the one hand, we enforce the decision method to perform a restock if both customers arrive simultaneously. On the other hand, the demand can sometimes be predicted quite well, such that restocks can often be postponed to a later time. As no costs are charged for restocking too early, we define the following errors:

False Positive
A restock which was not necessary that day, i.e. the inventory level would not have dropped below 0.

False Negative
No restock was performed while it was necessary that day, i.e. a bulk pick which could have been prevented.

A long horizon ($\tau = 500$) is simulated over several runs, where identical demand realizations are used per run.[1] Using this, we are able to compare several different deciders. For the Policy-s deciders, we test policy values of $s = 0, s = 1$ and $s = 2$, with $S = 2$ in all cases. The next decider is the (default) stochastic program, and as a comparison we include the all-knowing decider. This last decider provides the restock amount, knowing the value of the upcoming demand. Theoretically, this all-knowing decider can be used to evaluate how much one should be willing to pay to know the exact future.

[1]Each simulation run has different demand realizations, but the deciders at each run are compared with the same demand realizations.

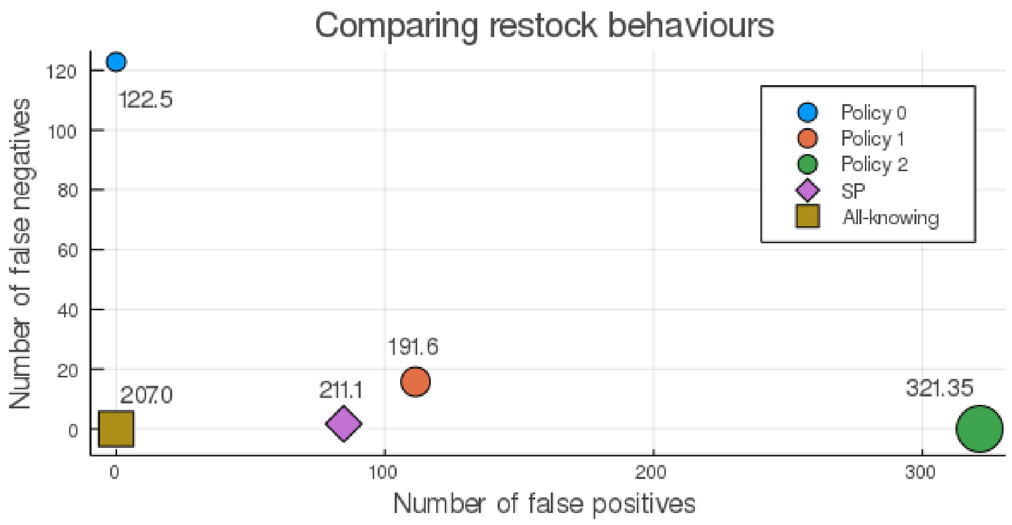


Figure 6.4: A comparison of the number of false positives and negatives for different deciders. The size and annotation of each marker indicates the total (average) number of restocks performed; note that a larger node indicates more restocks.

In Figure 6.4, observe the three deciders which behave as expected: the Policy-0 never has an unnecessary restock (i.e. no false positives), the Policy-2 always prevents bulk pick orders (i.e. no false negatives), and the all-knowing decider makes no mistakes. This in turn gives a clear indication of the number of restocks: the Policy-0 performs the lowest number of restocks, and the Policy-2 performs the most. The all-knowing decider performs on average 207 restocks, which is in line with the horizon of $\tau = 500$ and a daily expected demand of $\mathbb{E}[D] = 4/5$ (as a restock usually adds one or two items). In fact, the stochastic program performs only slightly more restocks than the all-knowing decider, whereas the Policy-1 method does not perform enough restocks. This can be seen clearly as the stochastic program has almost no false negatives, while the Policy-1 has several.

Concluding from the figure above, the two most interesting deciders are the Policy-1 and the stochastic program (besides the all-knowing decider); this is no surprise, as the optimal base stock policy value (as found by using stochastic programming) is equal to 1. For each of the used simulation runs, the actual numbers of false positives and negatives are plotted in Figure 6.5. Here, it is clear that the stochastic program always outperforms the policy method regarding both errors. Unfortunately, this comes at the cost of additional computational time required: each horizon of length $\tau = 500$ takes about 5 minutes to determine the restocks, compared to less than 0.3 seconds for a policy decider.

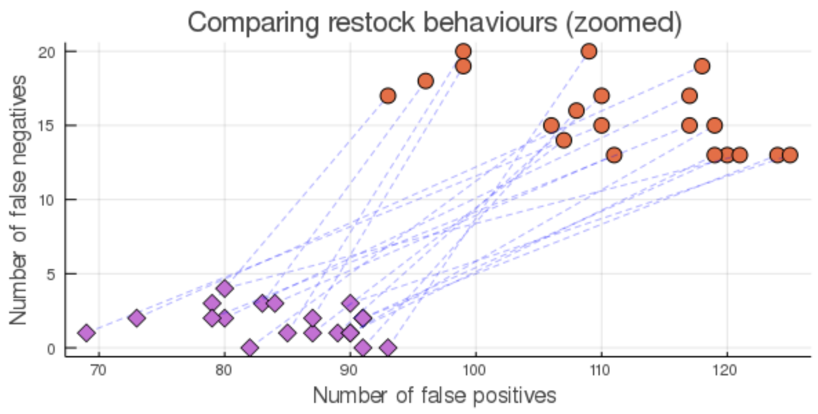


Figure 6.5: A zoomed view of the policy 1 and stochastic program deciders. Each node represents a simulation result, where diamonds are from the stochastic program, and circles from the policy. Each (dotted) line indicates which values result from the same demand realizations.

6.3 Example: Optimizing Employees

The second example which we investigate uses the employee module. We wish to show how a (standard) base-stock policy has difficulties coping with correctly using its staff, while the stochastic programming decider can adapt to the situation. The idea is that an ArrivalProcess is used with two types of customers: one type of customer γ_i arrives (almost) daily, but has a very small order amount of a single item; the second customer type γ^* arrives only once a week, yet orders quite large amounts from all items. As such, the pool of customers Γ for the ArrivalProcess is defined as follows:

$$\Gamma = (\gamma_1, \gamma_2, \gamma_3, \gamma^*) \tag{6.2}$$

$$\gamma_k = \begin{cases} A \sim \mathscr{U}[1,2] \\ O(i) = \mathbb{I}_{\{k=i\}} \end{cases} \qquad \forall k \in \{1,2,3\} \tag{6.3}$$

$$\gamma^* = \begin{cases} A \sim \mathscr{U}[4,6] \\ O(i) = 6 \quad \forall i \in \{1,2,3\} \end{cases} \tag{6.4}$$

For the warehouse, the most important chosen parameters are the number of items $N = 3$, the maximum inventory $V_i = 7$ for $i = 1,2,3$, and the costs for staff- and flex workers $15, 45$ respectively. By construction, all items should be (almost) fully restocked when γ^* arrives, as it empties all inventories almost completely.

When comparing the stochastic program decider, we chose to compare it to the 'optimal' policy, where the restock threshold $s = 1$ and the number of staff workers $\sigma = 0$. This policy turns out to be the best possible choice, found by simply comparing the (expected) costs for all different values of s and σ. For 30 separate runs, the total costs over a horizon of $\tau = 20$ are compared for these deciders, where the same (series of) demand realizations are compared per run.

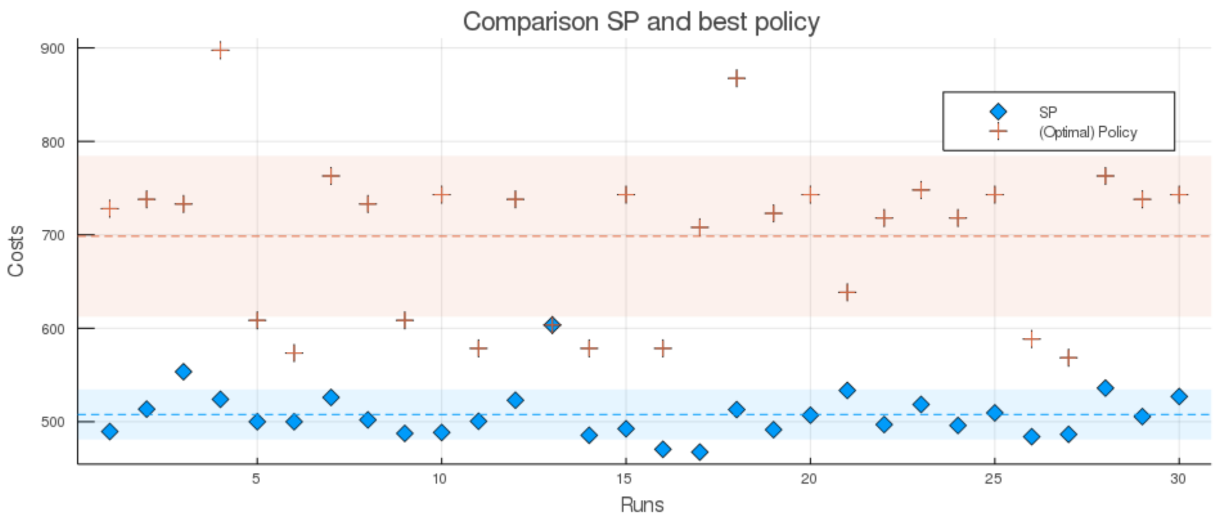


Figure 6.6: A comparison of the stochastic program and the (optimal) policy. The horizontal line and ribbon indicate the mean and standard deviation of the costs for both deciders.

Observe that in Figure 6.6, the stochastic program always results in lower costs than the policy does. For one particular run, we will investigate where this difference in costs originates from. The staffing levels of the stochastic program and policy are $\sigma = 1$ and $\sigma = 0$ respectively.

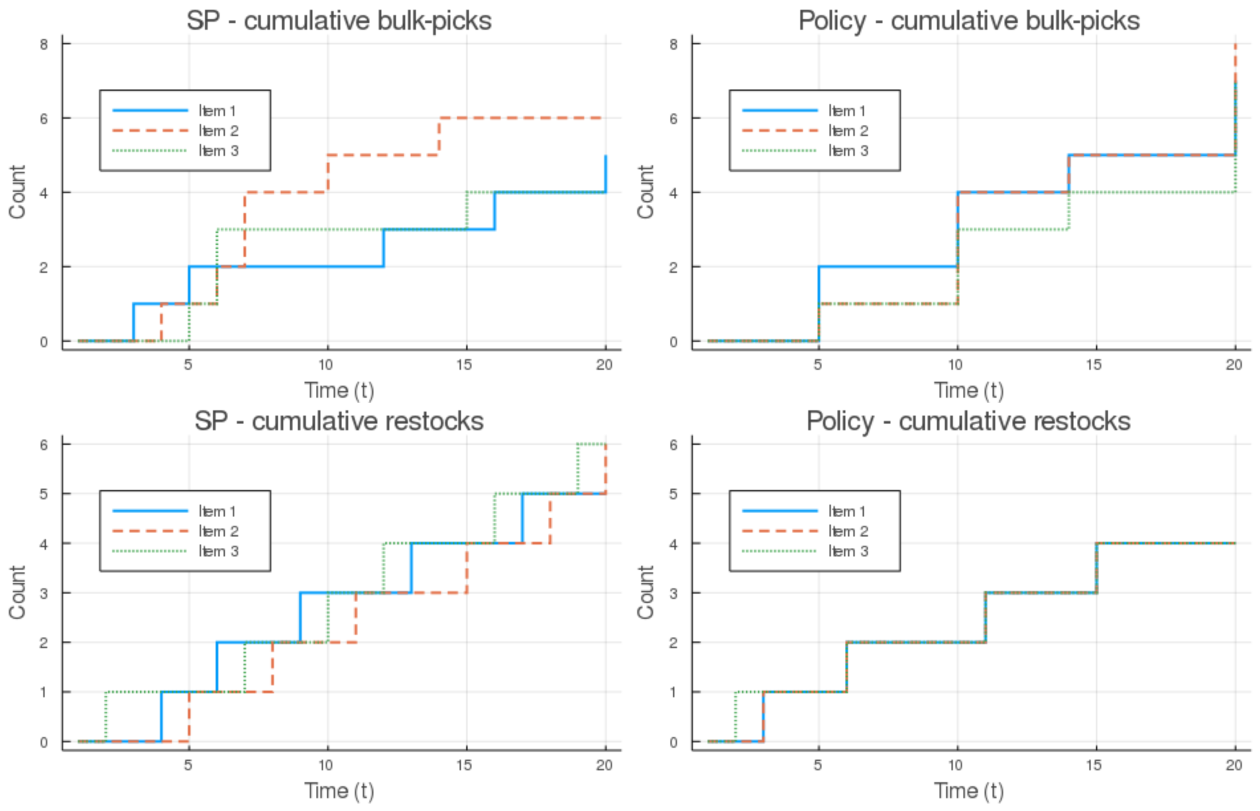


Figure 6.7: The behaviour of bulk-picks and restocks for both deciders.

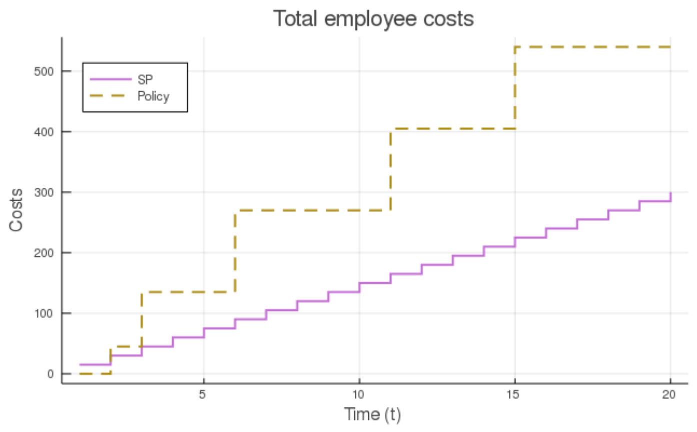


Figure 6.8: The behaviour of employee costs for both deciders. The policy has no (fixed) staffing costs, whereas the stochastic program never hires flex workers.

As seen in Figure 6.7, the total number of bulk-picks required appears to be similar for the stochastic program and the policy decider. However, the behaviour of the bulk-picks is quite different: the stochastic program has a few bulk-picks each day, but the policy results in a large amount of bulk-picks at particular days. The behaviour of restocks clearly shows that the stochastic program performs a restock nearly every day. The policy, however, only restocks the day after the arrival of γ^* (i.e. the day that all bulk-picks are performed). As such, the stochastic program is able to stock up on inventory before the arrival of γ^*, while the policy does not. Furthermore, the stochastic program ensures that no additional flex workers need to be hired to perform all restocks, while the policy only uses flex workers. Figure 6.8 shows the costs resulting from hiring workers, which clearly shows a difference between the deciders.

6.4 Case Study: Garbage Collection

The primary part of Chapter 6 focuses on the garbage collection problem, as mentioned in Section 1.2. Here, the warehouse model is slightly altered (in terms of definitions) to simulate an inventory control problem considering the collection of garbage in a municipality. The goal is to timely empty containers by using garbage trucks, such that no container ever exceeds its capacity. Of course, this has to be done by using the minimum amount of garbage trucks possible, as the costs should be minimized.

In this section, we will first describe the changes made to the warehouse model (Section 6.4.1). Next, we provide an overview of the parameters and deciders used (Section 6.4.2), and lastly indicate more in-depth results on the behaviour of these deciders in Section 6.4.3.

6.4.1 Altering the Warehouse Problem

The garbage collection problem is in fact quite similar to the warehouse model, as described in Chapter 3. The following alterations in definitions are used:

Containers $\Leftarrow$ items
Each container in the municipality is represented by a separate item type in the warehouse.

Container fill rate $\Leftarrow$ demand model
The containers are filled with garbage at a particular rate, which can be simulated using a demand model.

Garbage $\Leftarrow$ inventory
Instead of directly translating the inventory level to the amount of garbage, we instead translate the inventory level of a product to the amount of *air* in a container, as the amount of air decreases as garbage is added to the container.

Emptying containers $\Leftarrow$ restock
Containers should be emptied (i.e. refilled with air) to prevent overfull containers (i.e. there is no more air left). As such, a bulk-pick is a penalty which has to be paid if this does happen.

Garbage truck $\Leftarrow$ staff
Each garbage truck can only empty a certain amount of containers each day. This amount is constrained by either the amount of distance which needs to be travelled, or by the capacity of the truck. The labour amount can be chosen in such a way that it represents the minimum of both constraints.

Note that, in order to model the garbage collection problem, the routing aspect is ignored, which is a major simplification to the problem. Instead, we simply investigate which containers should be restocked, and assume that these containers can be reached.

6.4.2 Parameters and Deciders Used

The model parameters and settings are chosen with two particular ideas in mind: on the one hand, we wish to create a situation which resembles an actual garbage collection problem. On the other hand, some parameters are chosen differently, such that it can be made more clear how the stochastic program is able to perform differently than the heuristic deciders. A list of the most important model parameters is as follows:

Number of items $N = 100$
To investigate how the deciders perform on a problem of a larger scale, we choose a significantly higher number of items than in prior simulations.

Simulation horizon $\tau = 7$

The reason why the horizon length is chosen to be quite short is because the staffing level is fixed over the entire horizon, and the decider is not allowed to hire additional flex workers. As such, the staff level is fixed for one week, after which it can be adjusted and fixed for another week.

Demand D_i **non-i.i.d. demand model**

Each container has a different fill rate, provided by a normal distribution with negative correlation (see Section 4.2.2 for details). The reason for this is that there is some anti-correlation in the fill rate of a container: if a large number of citizens deposit their garbage today, they probably will not make a deposit again tomorrow.

Labour amount $L = 2$

Even though it is not realistic that each garbage truck can only empty 2 containers on a day, the labour amount is set quite low on purpose. The reason for this is that it allows more variation in the number of staff members required.

Maximum inventory $V_i = 1000$

While this inventory level (i.e. the amount of air) is measured in some arbitrary, unknown unit, it is only interesting what the demand is with respect to this maximum inventory level. For our case, an average demand (over all containers) of about 100 is taken, such that on average 10 of the 100 containers need to be restocked each day.

As the number of staff members is fixed over the horizon, and no flex workers can be hired, it does not make that much sense to use our default policy decider. As such, we use an alternative heuristic policy decider: given that K containers can be emptied on a day, empty the K containers with the (current) highest amount of garbage in them. If desired, a slight alteration can be made by emptying the K containers with the highest predicted amount of garbage by the end of the day (i.e. adding the expected demand to the current garbage level). In our case, K is determined easily by

$$K = L \cdot \sigma = 2\sigma \tag{6.5}$$

Besides this alternative policy decider, the default stochastic program is used with two particular settings: the first is that we investigate how well the stochastic program is in deciding what the staffing level would need to be for the horizon (given some initial inventory levels). If the stochastic program is not asked to find this staffing level, then the stochastic program is simply initialized multiple times, each time with a different fixed staffing level. The second setting is that we differentiate between two particular scenario tree structures: one tree is quite deep but narrow, while the other is very wide but only a few layers deep. The tree depth T and the number of child nodes C_t used in both settings are

$$T^{(1)} = 7 \qquad C_t^{(1)} = [2, 2, 2, 1, 1, 1, 1] \tag{6.6}$$

$$T^{(2)} = 3 \qquad C_t^{(2)} = [5, 5, 5] \tag{6.7}$$

6.4.3 Results

The results from this section are based on the parameters and settings as described in the previous chapter. Besides that, we set a time limit of 1000 seconds for the MILP solver, as it might take a significant amount of time to find the actual global minimum, while getting close to the best solution is often good enough.

Randomized Initial Inventories

The first results we discuss are based on simulations with randomized initial inventory levels. Each of the simulation runs is independent of the other simulations; per simulation run, all deciders are tested on the same demand realizations.

One of the first results which we find is that the initialization time required for determining the staffing level often takes longer than the provided time limit. The reason for this is because we did not limit the solver to determine only the optimal staffing level, but also to determine all other decision variables resulting from its scenario tree. As a result, it has significantly more freedom than having a fixed staffing level, which is most probably why it has so much trouble finding the optimal staffing level. To circumvent this, we could alter the decision problem (or the solver) to merely determine the staffing level; unfortunately, due to time constraints, this is outside of the scope of this project. For the remainder of this section, we do in fact let the stochastic program (SP) optimize its own staffing level, because it takes even longer to simulate with multiple staffing levels.

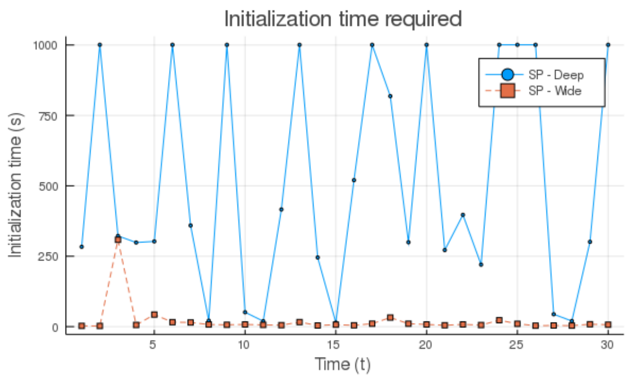


Figure 6.9: The time required for both stochastic programs to determine the staffing levels.

Next to this, we compared the different SP deciders with the policy deciders. The SP's were set to determine the optimal staffing level themselves, the policy deciders were simply run multiple times with multiple staffing levels. A quick overview of the average costs and computation times is given in Table 6.2.

Decider	**Staffing σ**	**Average costs**	**Average computation time (s)**
Stochastic Program - Deep tree	*	6360	3094
Stochastic Program - Wide tree	*	5875	60
Policy	4	> 40000	< 0.01
Policy	5	5927	< 0.01
Policy	6	7724	< 0.01

Table 6.2: Average costs and computation times for compared deciders. No difference is made between the two policy decider settings, as both resulted in similar results.

From these initial results, the SP with a wide but shallow tree appears to perform the best: it has reasonable computation time, and attains the best results on average. In fact, when investigating the restock behaviour considering false positives and negatives (as described in Section 6.2), this SP performs much less errors than the other deciders. As seen in Figure 6.10, the wide SP has strictly less errors than both the deep SP and the policy decider for multiple different simulation runs.

The stochastic program with the deep scenario tree does not seem to perform any better than the policy decider considering costs and restock errors, even though it does require significantly more computation time. This is likely because of the independence between simulation runs: any long-term costs (which the policy and wide tree might not prevent) are not simulated in the short horizon of only 7 steps. Next to that, deciding 7 steps in advance what to do might be too far ahead for what is needed, as it is primarily important to look far ahead when deciding the staffing level.

The policy deciders perform quite well considering average costs (except for the fixed staffing level of only 4), while maintaining an incredibly fast computation time. In fact, the average costs resulting from this policy approach are only slightly higher than the costs resulting from the wide SP. Because of its heuristic approach of restocking the fullest containers (i.e. with the lowest level of air), it has only a small amount of false negative restocks (i.e. an overfull container). However, this comes at the cost that if often has a significant number of false positive restocks, even on quite a small horizon. No direct costs result from restocking too early, but it is an indication that it could probably perform similarly using less staff members.

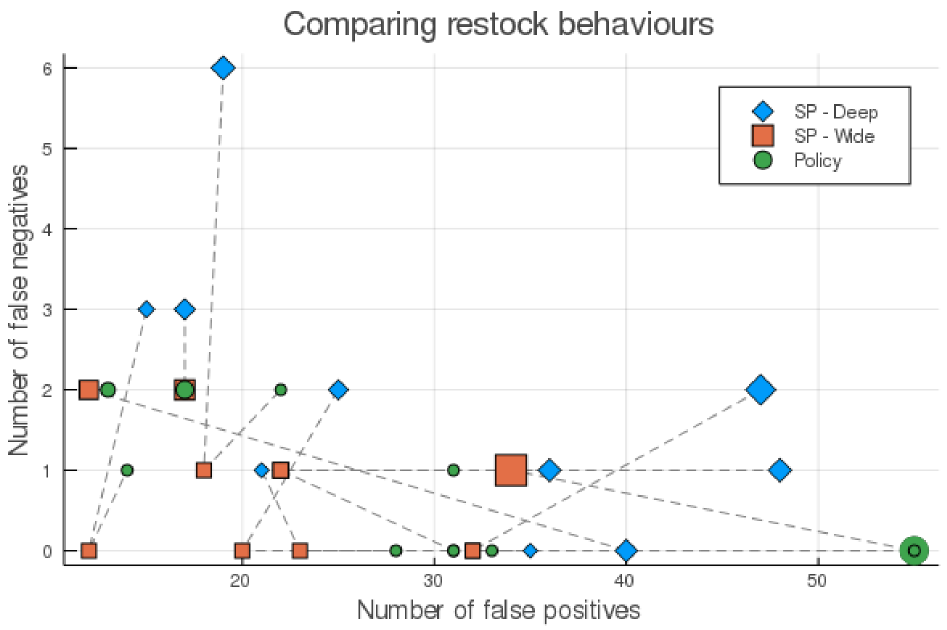


Figure 6.10: The number of false positives and negatives for several compared deciders, indicated for multiple simulation runs. The size of each marker indicates the total costs of that simulation run, where a smaller node indicates lower costs.

Continuing Inventory Levels

As concluded in the previous section, the stochastic program with a wide but shallow scenario tree is able to perform the best short-term decisions, and is not faced with higher costs in the long run. To circumvent this, we create a setting of dependent simulations: at each simulation of $\tau = 7$ steps (or 1 week), the staffing level per decider is fixed. After the horizon, the inventory level of the last day is copied to the next week, and a new simulation run is started. This way, the SP's are allowed to choose a new staffing level each week, but are in fact forced to deal with long-term effects of their (short-term) choices. The policy deciders are fixed with a staffing level of either 5 or 6 (as a staffing level of 4 turned out to be too little).

Figure 6.11 shows how both SP's decide how many staff members are required each week. Observe that the SP with a deep tree has almost no variance in the number of staff members required each week. The average number of staff members required is about 5.5, such that it usually sets the staff level to either 5 or 6. On a few occasions, it might need 7 trucks, probably because too many containers are too full. The SP with a wide but shallow tree, however, has a much higher variance in the number of staff members hired. One week it decides that only 4 trucks are needed, but the next week 11 trucks are needed to compensate for this. On average, it hires about 6.2 staff members per week, which is a lot higher than what the other SP hires.

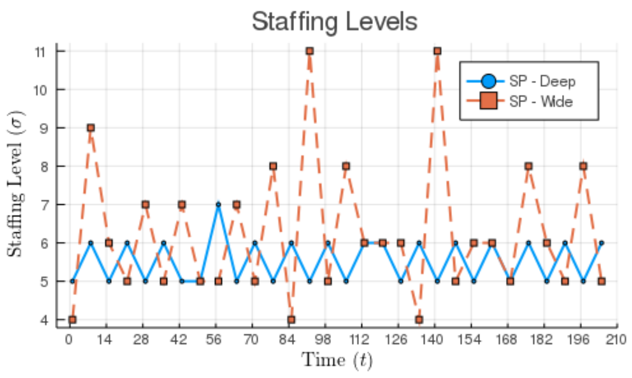


Figure 6.11: The staffing levels per week as initialized by the stochastic programs.

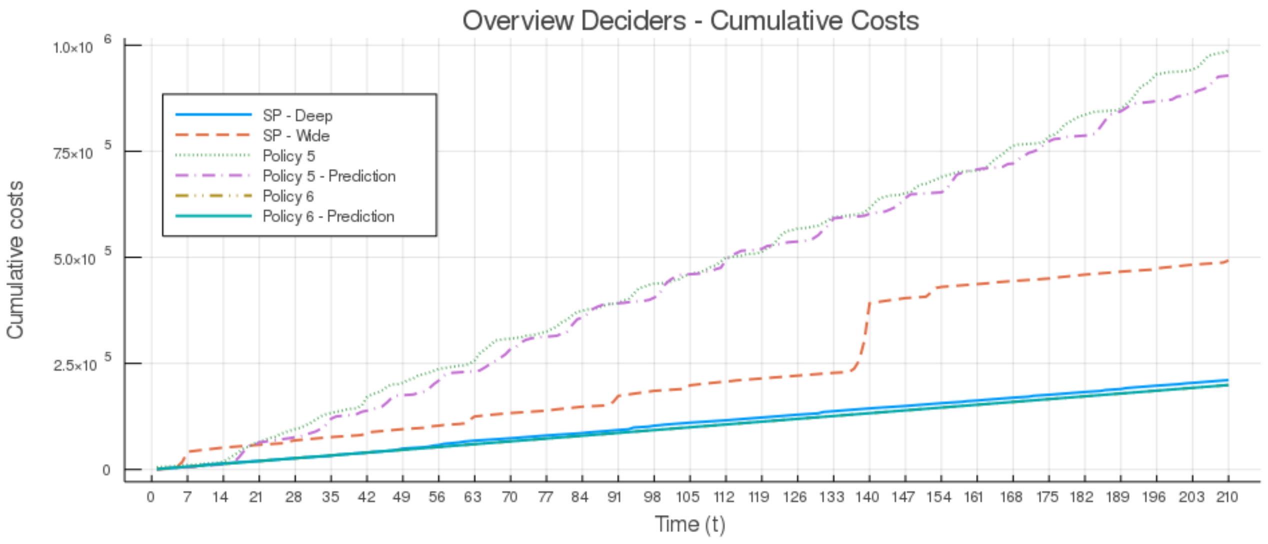
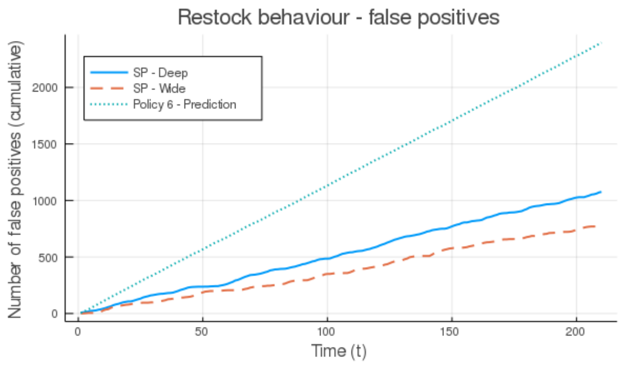


Figure 6.12: An overview of the cumulative costs of the compared deciders for 30 successive simulation runs.

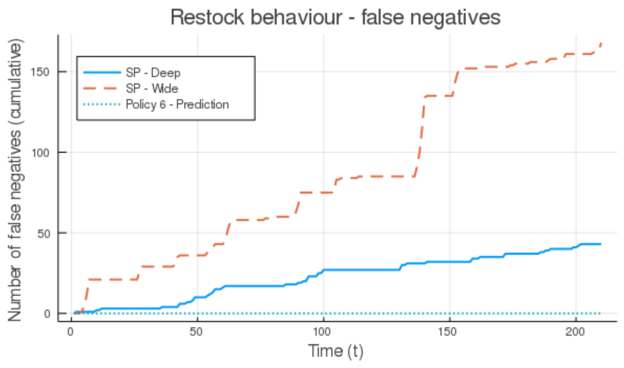
From Figure 6.12, it can be seen that the SP with the wide tree has trouble choosing the correct number of staff members: in particular, in the week of time 133 to 139, the costs rocket up due to bulk-picks at the end of the week. As the depth of the tree is only 3 steps deep, it did not take this into account when deciding the staffing level for that week. As such, it is important to realized that, even though a wide scenario tree might result in better results, a deep scenario tree must be used with respect to long-term decisions.

The highest costs originate from the policy deciders with a fixed staffing level of only 5. This is due to the fact that 5 trucks are not enough to restock all containers in time, resulting in minor bulk-pick costs each week. The lowest costs result from the deep SP and the policy deciders with a fixed staffing level of 6. The total costs are nearly the same, yet the policy deciders have a slight edge over the SP. The reason for this is twofold: the costs for hiring employees is (relatively) low (i.e. 150 per staff member per day), and the deciders are not penalized when restocks are performed too soon. Because of this, the SP attempts to minimize the number of staff members for only minor improvements in expected costs, which sometimes results in having too little staff to restock all containers.

When investigating the number of false positives over time (as shown in Figure 6.13a), it can be seen clearly that the policy method has significantly more unnecessary restocks than both SP's. The reason for this is that the staffing level is fixed at 6, such that the same number of restocks are performed every week. As seen in the staffing level behaviour of the SP's (Figure 6.11), the staffing level of 6 is not required every week. From the SP's, it can be seen that the wide scenario tree results in less false positives, primarily because it is often not able to prevent a bulk-pick. This can be seen clearly in Figure 6.13b.



(a) False positives.



(b) False negatives.

Figure 6.13: The number of cumulative (restock) errors for the compared deciders.

6.4.4 Concluding Remarks

Depending on what aspects are important, different settings can be chosen for the decider. If computation time is not an issue, it would be optimal to use a stochastic program where the scenario tree is both wide and deep. However, for our current settings, the standard policy methods are good enough, primarily because false positive restocks are not penalized, and the computation time required is almost non-existent.

Of course, we could be able to tweak the settings a bit more than we are currently able to. For example, it would make sense to initialize the staffing level with a deep and large tree, but afterwards use a much more shallow tree (or even a simple heuristic policy approach) to determine the daily restocks. With only minor adjustments to the program, such settings could be realized.

7 | Conclusion

In this thesis, we have created a simulation environment which can be used as a framework for several different optimization instances. With a focus primarily on inventory control in a warehouse, we have investigated several different techniques, and compared them in terms of computation time, accuracy and efficiency. The remainder of this chapter is focused on providing the concluding remarks based on the results (Chapter 6), and to give an overview of interesting future research.

7.1 Conclusion

First and foremost, the method of stochastic programming does not seem to be worth the additional computation time for our investigated examples. In the current setting, the objective value is only slightly better, while the computation costs skyrocket for larger scenario trees. Unfortunately, these larger scenario trees are in fact required. If a small or narrow scenario tree is used, the results are often relatively worse than a simple (heuristic) policy approach. Even when adding several additional complexities (such as staffing levels), the heuristics can be tweaked and updated, such that the stochastic program loses its benefit of finding better solutions.

Tweaking and updating heuristic algorithms might result in a proper solution, but this does not always result in a desirable situation. If adjustments need to be made iteratively to a simple heuristic, the final algorithm might not be as readable or easily implementable as desired. By creating exceptions on top of exceptions, the simple heuristics might not be as simple any more. For stochastic programming, however, the modular structure can be used much more easily. With respect to the linear program, we can add some constraints or variables, and the remainder of the algorithm remains exactly the same. In addition, we can easily use the multiple dispatch property of Julia: for example, we could define a new Decider type, and simply define new methods (with existing function names) for this particular Decider type. This way, no adjustments need to be made to the existing code, while we are able to add additional features.

Instead of iteratively changing a heuristic (policy) approach, an alternative would be to use a stochastic programming approach for the more significant one-time decisions, such as determining a restock threshold or staffing level. The scenario tree we use for the stochastic program can be quite large, in particular as it needs to be solved only once. By using a scenario tree which is both wide and deep, we are able to incorporate a large horizon for the decision, while including a very wide range of scenarios. Even when incorporating all sorts of additional complexities, we are able to find (near) optimal values, which can be used in the (heuristic) algorithm.

In conclusion, stochastic programming could be used to improve existing heuristics, in particular to determine the values used in heuristic algorithms. In some situations, using stochastic programming to directly determine decisions could be worthwhile: if the computational time remains within feasible bounds, it could be used to determine daily restock actions. By tweaking with parameters such as bulk-pick and restock costs, the timing of these restocks can be altered; one could restock such that a bulk-pick can (almost) never occur, or restock only when the odds are very slim that the current inventory level is enough. In any case, stochastic programming is a promising method when one wishes to incorporate uncertainty in its optimization process.

7.2 Future Research

There are quite some aspects which we were not able to fully research, because it was outside of the focus of this thesis. A list of the major aspects which could be researched further are described below.

Robust optimization
Even though we concluded from initial results that robust optimization did not provide valuable results, it might solve current issues we have regarding the computational time required. By further investigating the use of uncertainty sets, we might be able to have less conservative solutions, while maintaining proper solutions.

Progressive hedging algorithm
Another method aimed at decreasing the computational time required in the stochastic program. This algorithm could be used in combination with the current linear program solver, as this should result in much faster convergence rates.

Bender's decomposition
This technique can be used when solving large linear programs (as a result from the scenario tree) by creating a particular block structure of constraints and variables. By doing this, the decision variables can be solved in two successive stages, which greatly speeds up the computation time.

Extending the demand models
Currently, when we do not want to use a simple i.i.d. demand model, we are quite limited in how correlation is enforced. More time could be spend on determining new dependency enforcers, or the arrival process could be extended further by using dependent inter-arrival distributions. Next to that, other demand models could be constructed, which might be a much better fit to what happens in reality.

Modify restock and demand behaviour
As described in the assumptions, we currently model restocks and demand on a daily base, but this could be adjusted significantly. For example, instead of allowing a restock only at the end of the day, we could allow restocking to be performed throughout the day, where demand is also simulated over the course of a day. This way, additional complexities arise where staff members could be either restocking or picking orders, and the time penalties needed to perform these actions becomes even more important.

Uncertain lead times
As described in the assumptions, all decisions have zero lead time. By setting these lead times to be non-negative (stochastic) value, the difficulty of finding solutions increases tremendously. For example, in our garbage truck model, the time needed to restock a container is fixed. However, it might be that the roads are busier than expected, such that it would take much longer to empty a container.

Imperfect state information
Another assumption which could be removed is the usage of perfection information in the state descriptions. Continuing in our garbage routing problem, it is possible that we have no clear information regarding the amount of garbage in each container. Here, several different levels of imperfect information can be used. These levels range from an inventory sensor which is a few percent off to a complete black-box structure where no state information is known at all. When dealing with this, an educated guess should be made considering what the inventory levels would be, after which the remainder of the optimization is performed.

Bayesian statistics
While not investigated in this thesis, an interesting aspect would be if the predicted demand is not from the same demand model as the (simulated) uncertainty. For example, the parameters of the distribution

could be slightly off, or the dependency could be neglected completely. With this in mind, two new questions come to mind. Firstly, how well do the deciders perform when their demand model is incorrect? And secondly, how can we use Bayesian statistics to iteratively update the demand model settings?

Bibliography

[Beltran-Royol et al., 2010] Beltran-Royol, C., Escudero, F., and Rodriguez-Ravines, R. E. (2010). Multi-stage stochastic linear programming: Scenarios versus events.

[Ben-Tal et al., 2009] Ben-Tal, A., El Ghaoui, L., and Nemirovski, A. (2009). *Robust optimization*, volume 28. Princeton University Press.

[Bezanson et al., 2017] Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98.

[Boland et al., 2008] Boland, N., Dumitrescu, I., and Froyland, G. (2008). A multistage stochastic programming approach to open pit mine production scheduling with uncertain geology. *Optimization online*, pages 1–33.

[Cuadras, 2002] Cuadras, C. (2002). On the covariance between functions. *Journal of Multivariate Analysis*, 81(1):19 – 27.

[Defourny et al., 2011] Defourny, B., Ernst, D., and Wehenkel, L. (2011). Multistage stochastic programming: A scenario tree based approach to planning under uncertainty. *LE, Sucar, EF, Morales, and J., Hoey (Eds.), Decision Theory Models for Applications in Artificial Intelligence: Concepts and Solutions. Hershey, Pennsylvania, USA: Information Science Publishing.*

[Dunning et al., 2017] Dunning, I., Huchette, J., and Lubin, M. (2017). Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320.

[Edgeworth, 1888] Edgeworth, F. Y. (1888). The mathematical theory of banking. *Journal of the Royal Statistical Society*, 51(1):113–127.

[Gorissen et al., 2015] Gorissen, B. L., Yanıkoğlu, İ., and den Hertog, D. (2015). A practical guide to robust optimization. *Omega*, 53:124–137.

[Gurobi Optimization, 2019] Gurobi Optimization, L. (2019). Gurobi optimizer reference manual.

[Kim et al., 2015] Kim, G., Wu, K., and Huang, E. (2015). Optimal inventory control in a multi-period newsvendor problem with non-stationary demand. *Advanced Engineering Informatics*, 29(1):139 – 145.

[Rockafellar and Wets, 1991] Rockafellar, R. T. and Wets, R. J.-B. (1991). Scenarios and policy aggregation in optimization under uncertainty. *Mathematics of operations research*, 16(1):119–147.

[Sethi and Cheng, 1997] Sethi, S. P. and Cheng, F. (1997). Optimality of (s, s) policies in inventory models with markovian demand. *Operations Research*, 45(6):931–939.

[Shapiro and Philpott, 2007] Shapiro, A. and Philpott, A. (2007). A tutorial on stochastic programming. *Manuscript. Available at www2. isye. gatech. edu/ashapiro/publications. html*, 17.

[Watson and Woodruff, 2011] Watson, J.-P. and Woodruff, D. L. (2011). Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems. *Computational Management Science*, 8(4):355–370.

A | Appendix

A.1 Glossary

Item A single package (or product) which is ordered by a customer, and must be order-picked from one of the picking areas.

Inventory The number of items in stock in a particular area.

Bulk-pick area Large area of a warehouse where products are stored in bulk.

Direct-pick area Smaller area of a warehouse where products can easily be order-picked.

Demand The uncertainty in the model, indicating the number of products ordered by the customers.

Restock The action of moving items from the bulk-pick area to the direct-pick area.

Stochastic programming An optimization technique, based on discretizing uncertainty, constructing a scenario tree, and optimizing a linear program over all scenarios.

Robust optimization An optimization technique, aimed at finding an optimal solution given an uncertainty set. The provided solution is found by optimizing over the worst-case scenario(s) in the uncertainty set.

Scenario tree A discretization of the uncertainty, where different scenarios are grouped in a tree-like structure.

Nonanticipativity constraints Used in stochastic programming, these indicate that decisions are independent of upcoming uncertainty, i.e. all decisions with the same past must be equal.

Receding horizon A technique used in the simulation of a long horizon; only a small part of the horizon is focused on, and this focus is shifted with small steps in successive simulation steps.

Base stock policy A heuristic approach commonly used in inventory control problems. A restock threshold is defined, such that products are restocked if the inventory is below the threshold.

Big-M constraints Used only in the formulation of constraints and variables in a linear program. By using an upper bound M, an if-else constraint can be formulated.

Correlation The Pearson correlation coefficient, indicating the linear correlation between two random variables. It is defined on the interval $[-1, +1]$, where $+1$ indicates full correlation, 0 indicates no correlation, and -1 full anti-correlation.

Multiple Dispatch Used in the Julia implementation. It is the term used to describe that methods with the same (function) name are constructed, where each method requires different inputs. Each method can be completely different from the other methods with the same name.

A.1.1 Abbreviations

FIFO First In First Out policy.

MILP Mixed Integer Linear Program.

SP Stochastic Program.

I.I.D. Independent and Identically Distributed.

A.2 Progressive Hedging Algorithm

Algorithm 1 Progressive Hedging

```
 1: Initialize $\nu = 1$, notTerminated $= true$. Fix $r > 0, \varepsilon > 0$
 2: for $\sigma \in \mathcal{S}$ do
 3:     Initialize $\Psi^0(\sigma) \equiv 0$
 4:     Find scenario numbers $J_\sigma$                                  ▷ Not used in pseudocode
 5:     $X^0(\sigma) \leftarrow$ optimal decisions for (solved!) SP corresponding to scenario $\sigma$ without NA constraints
 6: end for
 7: Initialize $\hat{X}^0 \leftarrow \sum_{\sigma \in \mathcal{S}} \mathbb{P}(\sigma) \cdot X^0(\sigma)$                ▷ $\mathbb{P}(\sigma) = (S_T)^{-1}$
 8: while notTerminated do
 9:     for $\sigma \in \mathcal{S}$ do
10:         Construct SP corresponding to scenario $\sigma$ without NA constraints, $X^\nu(\sigma)$ are decision variables
11:         Add $\Psi^{\nu-1}(\sigma) \cdot X^\nu(\sigma) + \frac{1}{2} \cdot r \cdot \left(X^\nu(\sigma) - \hat{X}^{\nu-1}\right)^2$ to the objective value of SP
12:         Optimize the model, store optimal values $X^\nu(\sigma)$
13:     end for
14:     $\hat{X}^\nu \leftarrow \sum_{\sigma \in \mathcal{S}} \mathbb{P}(\sigma) \cdot X^\nu(\sigma)$
15:     for $\sigma \in \mathcal{S}$ do $\Psi^\nu(\sigma) \leftarrow \Psi^{\nu-1}(\sigma) + r \cdot \left(X^\nu(\sigma) - \hat{X}^\nu(\sigma)\right)$
16:     if stopping criterion $< \varepsilon$ OR $\nu$ too large do notTerminated $= false$
17:     $\nu \leftarrow \nu + 1$
18: end while
```