Reinforcement Learning and Artificial Intelligence in the context of revenue management

Freie wissenschaftliche Arbeit
zur Erlangung des akademischen Grades
"Master of Science"
Studiengang: Finanz- und Informationsmanagement

an der
Wirtschaftswissenschaftlichen Fakultät
der Universität Augsburg

– Lehrstuhl für Analytics & Optimization –

Eingereicht bei:	Prof. Dr. Robert Klein
Betreuer:	Dr. Sebastian Koch
Vorgelegt von:	Stefan Glogger
Adresse:	Riedstr. 21
	86647 Buttenwiesen
Matrikel-Nr.:	1471113
E-Mail:	stefan.glogger@student.uni-augsburg.de
Ort, Datum:	Augsburg, 30. September 2019

Contents

List of Figures

List of Tables

Chapter 1

Introduction

…briefly explains how…

the

This document presents my Master's Thesis for the M. Sc. Programme *Finance and Information Management*, in which I explain, implement and compare performance of various techniques and algorithms in revenue management. This first chapter gives a short overview of the history of revenue management, introduces the problem to be discussed in the following chapters and puts the thesis in context of different fields of revenue management. It furthermore gives a short overview of the chapters to come and briefly elaborates on how I see this thesis fitting in the context of my study programmes.

The concept of revenue management originated as a result of the U. S. Airline Deregulation Act of 1978. Prior to this, prices had been strictly controlled and standardized by the U. S. Civil Aviation Board (CAB). Afterwards, airlines were free to choose prices, change schedules and alter offered services at their will, without CAB approval. This deregulation resulted in an immediate appearance of low-cost/no-frill airlines. By operating on lower labour costs and less service on board, profitable fares up to 70% lower than the ones of major carriers could be offered and market pressure on classical airlines increased. All airline companies now faced the problem of which price to offer at any given circumstances.

providing

The traditional, rather simple assumption of independent demand didn't work. It expects demand to be independent of the current market conditions such as prices offered by competitors, (day-)time of departure, frequency of departures, brand image or others, to be found in e. g. Talluri and Ryzin (2005). Furthermore, low fare demand is generally expected to come before high fare demand, as business travellers prefer the flexibility to possibly change their schedule. Thus, the problem reduced to: How much capacity should be reserved for the high fare demand, that appears later? But as the purchase of a ticket clearly depends on the market situation (e. g. an exceptional Formula 1 race at one specific weekend) or the continuing expansion of low-cost airlines offering undifferentiated fare

day/time

besser: Now all the airline companies faced…

entweder: …now also allows you to account for…
oder: …now also accounts for…

independence

structures, this simple assumption of independent falls short in practice as also stated in Bront et al. (2009).

As a result, academia shifted focus to more sophisticated models building on customer choice behaviour. Demand is assumed to depend on the currently offered products, which might change depending on the circumstances. This assumption now allows to also account for buy-up (buying a higher fare when lower fares are closed) and buy-down (switching to a lower fare when discounts are available). The assumption of customer choice is then incorporated in dynamic pricing models, as e. g. studied by Gallego and van Ryzin (1997), Bitran et al. (1998) or Feng and Gallego (2000).

e.g. as

The underlying problem is referred to as *capacity control under customer choice behaviour* and can be summarized as: Which products should be offered at any given circumstance? In the airline setting, circumstance is determined mainly by time to departure and remaining capacity. As many airlines operate so called *hub-and-spoke networks*, allowing them to offer service in many more markets than with point-to point connection, the problem becomes more challenging as stated e. g. in Talluri and Ryzin (2005). This network structure can be incorporated by modelling each point-to-point connection (single leg) by one separate resource and each seat as one unit of capacity. Furthermore, different customers (e. g. business vs. leisure traveller) can be distinguished by modelling them as different customer segments. Thus, the central problem in capacity control can be stated as:

until

> *In the setting of a network consisting of several flights (edges) connecting certain cities (vertices), when given a particular seat capacity (weight of edge) and a certain time to departure, which combination of products shall be offered to customers?*

intentionally done

Note that revenue management is relevant in many fields, even though we introduced the problems purely in the airline setting. The introduction was done intentionally like this as revenue management is closely connected to the airline industry and it increases total profitability by 4 to 5% of revenues Talluri and Ryzin (2005)[1]. However, revenue management appears in any decision situation related to demand management. Talluri and Ryzin (2005) structured those decisions in three basic categories:

- Structural decisions: How the selling process is organized (e. g. negotiations among individuals, auctions with restricted access, publicly posted prices) or how additional terms are structured (e. g. volume discounts or refund options).

[1] We want to point out that Soutwest Airlines is often mentioned as a counterexample. But even though its pricing structure is very simple, revenue management systems are still used as pointed out by Talluri and Ryzin (2005).

Southwest?

hier fehlt etwas z.B. …as discovered by…
oder …as pointed out by…

- Price decisions: How to set individual offer prices, reserve prices (in auctions) and posted prices or how to price distinct products over time.
- Quantity decisions: Whether an offer to buy should be accepted or rejected or when to withhold a product from the market in order to sell it at a later point in time.

For this thesis, structural and price decisions have been made already: Prices are fixed in advance and posted publicly. The quantity decisions remains and we can now formulate the goal of this thesis. We want to explore different methods of "solving" the capacity control problem under choice behaviour in a single-leg and multi-leg setting by implementing the algorithms, let them cope with exemplary test scenarios and compare their performance. Furthermore, new methods such as the usage of Neural Networks shall be discussed and evaluated.

remain

comparing their performances

remainder

The remaining of the thesis is structured as follows. ?? gives an overview of existing literature. Chapter 2 describes the problem formally and introduces various solution methods. Chapter 3 presents theory on how to compare different methods based on statistics. Chapter 4 evaluates the different methods in one single-leg and one multi-leg scenario. Finally, Chapter 5 summarizes the thesis and presents an outlook for future research. The code of the implementation is given in Appendix A.1 but can also be found on .

explain

Briefly, I want to elaborate on how I see this thesis fitting into my study programmes and my personal goals going along with it. This thesis shall combine the programme Finance and Information Management, with its interdisciplinary components of Business, Mathematics and Informatics, together with my personal interests and professional working attitude. The optimization component is covered by modelling the situation, creating an optimization problem, implementing this in software, visualizing the results and applying duality theory. The business component can be found in concepts like utility function, complementary assets and substitutes. Basic probability theory is incorporated with probability spaces, Kolmogorov-axioms, random variables and markov chains. Concepts of statistics are included as statistical tests are created, implemented and results presented. The informatics component is covered by preparing code in the modern[2] programming language Python and following modern software standards, as introduced by The Zen of Python, the style guide PEP 8 and reproducibility. The mathematical component can be found throughout the thesis by precise notation, reformulation of some concepts found in papers, and short mathematical components in footnotes or the appendix. The the-

such as the

[2]Python is heavily used in the Data Science community.

sis should be a neat document, properly set in LaTeX and showing my further enhanced typographic and layout skills, also in TikZ. Overall, this thesis should present a concise overview of techniques currently applied in revenue management and can be used as a starting ground for future research in this field.

Chapter 2

Problem and Approaches to Solution

This chapter introduces the classical revenue management problem of capacity control problem under customer choice behaviour as can be found e. g. in Koch (2017) or in Strauss et al. (2018). The problem consists of deciding which products to offer at a given point in time and Section 2.1 describes it in general as well as formulates it as a dynamic program. Various methods of solving the problem are laid out in Section 2.2, before Section 2.3 introduces a more enhanced solution method, Approximate Dynamic Programming.

2.1 The problem

This section describes the capacity control in general in Section 2.1.1, presents a concrete example in the airline setting in Section 2.1.2 and introduces the formulation of the resulting choice based capacity control model as a dynamic program in Section 2.1.3.

2.1.1 General description

We cover a setting that is most regularly seen in every-day life: A company offers a bundle of products to heterogeneous customers arriving over time. Let us now introduce some terminology. *Products* often correspond to services that have to be delivered at a given point in time. The period of time from now until delivery is referred to as *selling horizon* (or booking horizon), is considered finite and is split into a finite set of consecutive *time points*. The set of available products is also assumed to be finite and without loss of generality (w. l. o. g.) constant over time[1]. Each individual product is characterized by a

[1] If the set of available products change over time, let S_t denote the set of available products at time t and let $\mathcal{T}$ represent the finite set of time points. Then, $S \coloneqq \cup_{t \in \mathcal{T}} S_t$ is again finite (finite union of finite sets) and can be taken as the constant set of available products at each time point.

…of operating businesses…
oder …of operating companies…

given *price* and usage of certain, finitely many resources. These *resources* might be shared among different products.

besser: usually

The goal of business companies operating is to increase profits, which is equivalent to maximizing revenue. As stated, the price of individual products (leading to revenue) is assumed to be fixed, which is a realistic assumption for short terms considering potentially large costs coming with a change of price (e. g. selling products via catalogue). Furthermore, capacities are fixed in the short term (costly to increase production as e. g. more employees have to be hired) and are considered perishable. Often, capacity comes with high fixed costs and low marginal costs resulting from selling an additional product at any price (e. g. costs for operating one flight with 97 customers are mostly the same as for operating the same flight with 98 customers). Such a profit and cost structure justify the usage of revenue as a proxy for profit as also stated in Strauss et al. (2018).

justifies

life
into
precisely

Switching to the customer side, the consideration of a heterogeneous customer group allows for modelling most real world scenarios. Not every customer is the same, but it is certainly reasonable to combine individuals to groups sharing similar characteristics. Some characteristics will be more prevalent in the overall population (more precise, the group of people that represents all persons interested in the companies' products). Different customers have varying needs, thus different preferences and different willingness-to-pay, leading ultimately to different purchases depending on the currently offered set of products. Thus, the company can influence the sales process by altering the offered set of products over the booking horizon, which is exactly the task of capacity control under customer choice behaviour.

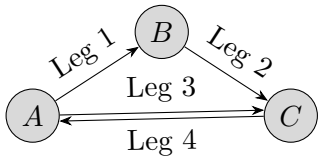
2.1.2 Description in airline setting

The concepts introduced above are now applied to a concrete airline setting, which will later be used to present the solution techniques. The airline company AirCo operates a flight network connecting cities A, B and C as depicted in Figure 2.1a. The booking horizon consists of 20 time steps (e. g. today is Monday, booking is possible four times every day until and including Friday and flights depart on Saturday). The products to be offered are summarized in Figure 2.1b. Note that two products are available on Leg 1, i. e. the expensive product 1 and the less expensive product 2 (e. g. due to separate business and economy classes). A sell of either product 1 or product 2 will result in the reduction of available capacity on Leg 1 by one unit. Furthermore, product 6 presents an option to get from city A to city C (via city B) and thus a sell of product 6 results in a reduction

sale

sale

(a) Airline network.



(b) Products.

Product	Origin-destination	Fare
1	$A \to B$	5
2	$A \to B$	3
3	$B \to C$	5
4	$A \to C$	12
5	$C \to A$	10
6	$A \to B \to C$	8

(c) Resources.

Leg	Capacity
1	8
2	4
3	4
4	8

(d) Customers.

Seg.	λ_l	Consideration tuple[a]	Preference vector	No purchase preference	Description
1	0.3	$(1, 2)$	$(4, 8)$	2	Price sensitive, (A→B)
2	0.2	$(1, 2)$	$(6, 5)$	2	Price insensitive, (A→B)
3	0.2	$(4, 6)$	$(8, 4)$	1	Price sensitive, fast (A→C)
4	0.1	(5)	(2)	2	Opportunity traveller, (C→A)

[a]Note that in contrast to Bront et al. (2009), we use the mathematically correct terminology as *tuple* does have an inherent order, while *set* does not (and thus makes no mathematical sense to be combined with a preference vector referring to the order of products in the set).

Figure 2.1: Working example to illustrate problem with $T = 20$ time periods.

of capacities on both Leg 1 and Leg 2 by one unit. Also product 4 is connecting city A and city C (on the direct flight) but is more expensive then product 6 (e. g. the company assumes the shorter travel time is of value to some travellers).

Furthermore, capacities are presented in Figure 2.5c, so there are eight available seats on Leg 1, but just four seats are available on Leg 2. Customers are characterized as stated in Figure 2.5d, so most customers belong to segment 1 (higher value of λ_l representing larger size of customer segment l), want to travel from city A to city B and prefer the budget option (as the value of preference for the latter product is much higher and a higher value corresponds to a higher preference). Customers belonging to segment 2 also want to travel from A to B, but are price insensitive, as they value the higher comfort of the more expensive product 1 just little higher as the cheaper product 2. Business customers that need to travel from A to C are represented by customer segment 3 as can be seen by the low preference for no-purchase. slightly more than

Symbol	Characteristic	Meaning
i, j, h, t, l, p	general small letter	index
n, m	specific small letters	maximum index
$[n]$	set of natural numbers	$\{1, \dots, n\}$
T, L	general capital letter	maximum index
S	specific capital letter	tuple of elements
x_1	small with index	single number
$\boldsymbol{x}$	small and bold	vector
$(x_1, \dots, x_n)$	parenthesis around single numbers	row vector
$(\dots)^\intercal$	$^\intercal$ on upper side	transpose

Table 2.1: Concepts of notation

2.1.3 Formulation as dynamic program

uses

Extensive literature deals with revenue management and came up with formal notation suitable for the analysis of capacity control models. Good overviews can be found in Klein and Steinhardt (2008) or in Phillips (2011). In this thesis, we introduce notation for a capacity control model first under a general discrete choice model of demand and then directly using a multinomial logit model of demand. Notation is closely linked to Koch (2017) and the standard representation is used for a better orientation with the main ideas outlined in Table 2.1 .

Formal notation

First, we present the situation with its inflexible components. Consider a firm that produces n distinguishable products indexed by $j = 1, \dots, n$. We introduce the notation $[n] := \{1, \dots, n\}$. Product j comes with a certain price leading to a revenue of r_j when the product is sold. The revenues are bundled by the vector $\boldsymbol{r} = (r_1, \dots, r_n)^\intercal$. A total of m distinct resources are used for production, which are indexed by $h \in [m]$. In order to produce one unit of product j, certain resources are needed and the consumption of resources is captured by the vector $\boldsymbol{a}_j = (a_{1j}, \dots, a_{mj})^\intercal$ are necessary, with $a_{hj} = 1$ if resource h is needed for production of product j and $a_{hj} = 0$ otherwise. The resource consumption can be aggregated in the incidence matrix $A = (a_{hj})_{h\in[m],j\in[n]} \in 0, 1^{m\times n}$. Additionally, the firm already identified the group of potential customers and segmented this group into a total of L customer segments, which are indexed by $l \in [L]$. A customer of segment l arrives with probability λ_l and this parameter can be adjusted such that it

…necessary resources is captured by…

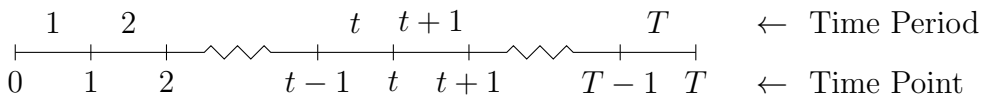


Figure 2.2: Visualization of booking horizon with its time periods vs. its time points.

twice
matches the reality (e. g. there are two times as many customers belonging to segment a as then there are of segment b, so we choose $\lambda_a = 2\lambda_b$) and the model assumptions (only one customer arrives during each period). .

The finite time period from when booking is possible for the first time to when it is possible for the last time is discretized into a total of T sufficiently small time periods, such that in each individual time period at most one customer arrives. Time periods are indexed by $t \in [T]$. To be very precise (particularly relevant for correct implementation), booking itself is just possible at the beginning of a time period, i. e. for time period t, booking is possible at time point $t-1$. Furthermore, one customer purchases at most one product.

Secondly Second, we look at the flexible components of the model, i. e. those components that might change over time. Initially, i. e. at the start of the first time period (time point 0), the available capacity of resource h is given by c_h^0 and thus all available capacities are described by the vector $\boldsymbol{c}^0 = (c_1^0, \ldots, c_m^0)^\intercal$.

Let us now explore what happens if product j is purchased during time period t, i. e. at time point $t-1$. The old capacity vector is given by $\boldsymbol{c}^{t-1}$. As product j claims capacities according to $\boldsymbol{a}_j$, the capacity vector reduces accordingly to $\boldsymbol{c}^t = \boldsymbol{c}^{t-1} - \boldsymbol{a}_j$. Time moves forward, such that the last selling might occur at time period T, i. e. at time point $T-1$.

The distinguishment of time period and time point is also visualized in Figure 2.2 and one more comment on this topic: Time period t can be seen as the following half open interval between time points: $(t-1, t]$.

To sum it up, the firm aims at increasing the value of the products sold and has flexibility in the sets offered. Thus, the decision variables for each time period $t \in [T]$ are given by $\boldsymbol{x}^t = (x_1^t, \ldots, x_n^t)^\intercal$ with $x_j^t = 1$ if product j is offered during time period t (i. e. chosen at time point $t-1$) and $x_j^t = 0$ otherwise. Thus, for each time period t and the prevalent capacity $\boldsymbol{c}$, the offer set $\boldsymbol{x}$ has to be determined. Note that in machine learning terms, this mapping is defined as a policy $\boldsymbol{\psi}(\boldsymbol{c}, t)$.

We can put the goal mentioned above in formulas after introducing the random variable X^t as the product which is purchased at time point t, with the notation $X^t = \{0\}$ if no-purchase is made at time point t. So we use $\Omega := \{0, 1, \ldots n\}$ as the underlying set (no-

purchase plus all purchasable products) and can use $\mathcal{F} = \mathrm{P}(\Omega)$ as underlying sigma algebra, with $\mathrm{P}(\cdot)$ describing the power set[2]. Furthermore, we introduce a probability measure $p_{\boldsymbol{x}^t}(\cdot)$ and use the shorthand $p_{\boldsymbol{x}^t}(j) = p_{\boldsymbol{x}^t}(\{j\}) = p_{\boldsymbol{x}^t}(X^t = \{j\})$ as the probability of product j being purchased at time point t when tuple $\boldsymbol{x}^t$ is offered. Note that p has to satisfy the corresponding Kolmogorov axioms, as it is a probability measure:

$$p_{\boldsymbol{x}^t}(j) \geq 0 \quad \forall j \in \{1, \dots n\} \qquad \text{(non-negativity)} \tag{2.1}$$

$$p_{\boldsymbol{x}^t}(\Omega) = 1 \qquad \text{(unitarity)} \tag{2.2}$$

$$p_{\boldsymbol{x}^t}(\cup_{j\in J}\{j\}) = \sum_{j\in J} p_{\boldsymbol{x}^t}(j) \quad \forall J \subset \Omega \qquad \text{(additivity in discrete setting)} \tag{2.3}$$

How to arrive at those probabilities is discussed in Section 2.1.3. Now, we formalize the aim of the firm. The expected revenue-to-go starting with time period t and capacity $\boldsymbol{c}$ is given by the value function $V(t, \boldsymbol{c})$, which maximizes the expected revenue and is mostly formulated recursively because of complicated evolutions due to capacities changing over time.

$$V(t, \boldsymbol{c}) = \max_{x^t \in \{0,1\}^n} \left\{ \sum_{j\in[n]} p_{\boldsymbol{x}^t}(j) \left(r_j + V(t+1, \boldsymbol{c} - \boldsymbol{a}_j)\right) + p_{\boldsymbol{x}^t}(0) V(t+1, \boldsymbol{c}) \right\} \tag{2.4}$$

$$= \max_{x^t \in \{0,1\}^n} \left\{ \sum_{j\in[n]} p_{\boldsymbol{x}^t}(j) \left(r_j - \Delta_j V(t+1, \boldsymbol{c})\right) \right\} + V(t+1, \boldsymbol{c}) \;\; \forall t \in [T], \boldsymbol{c} \geq 0 \tag{2.5}$$

with $\Delta_j V(t+1, \boldsymbol{c}) \coloneqq V(t+1, \boldsymbol{c}) - V(t+1, \boldsymbol{c} - \boldsymbol{a}_j)$ and boundary conditions $V(T+1, \boldsymbol{c}) = 0$ if $\boldsymbol{c} \geq \boldsymbol{0}$ and $V(t, \boldsymbol{c}) = -\infty \; \forall t \in [T+1]$ if $\boldsymbol{c} \ngeq \boldsymbol{0}$, i. e. $\exists h \in [m] : c_h < 0$.

Note that for determining the optimal offerset, the very last summand $V(t+1, \boldsymbol{c})$ is irrelevant for the maximization as it is constant over all offersets. The optimal offerset $\hat{\boldsymbol{x}}^t$ is thus given by

$$\hat{\boldsymbol{x}}^t = \operatorname*{arg\,max}_{x^t \in \{0,1\}^n} \left\{ \sum_{j\in[n]} p_{\boldsymbol{x}^t}(j) \left(r_j - \Delta_j V(t+1, \boldsymbol{c})\right) \right\} \;\; \forall t \in [T], \boldsymbol{c} \geq 0 \tag{2.6}$$

[2]Note that $\mathrm{P}(\Omega)$ is finite as Ω is finite by assumption

Discrete Choice Model for Customer Behaviour

We already discussed that at each time period at most one customer arrives (out of L customer segments), who can purchase at most one product. The only remaining question is which product is being purchased during a particular time period. There exist multiple models to describe purchase probabilities, but many of them are based on preferences in accordance with utility. We first introduce the utility concept in general and then introduce the multinomial logit model (MNL), which is well known in practice and can be found e. g. in Train (2009). Note that we introduce the concept for a general offerset $\boldsymbol{x}$ in order to easily focus on the important aspects. In reality, and in our implementation, the offerset is considered time-dependent $\boldsymbol{x}^t$.

Consider customer segment l. Let $u_{lj} \in \mathbb{R}_0^+ \ \forall j \in [n]$, resp. $u_{l0} \in \mathbb{R}^+$ denote the utility that a customer of segment l assigns to product j, resp. to the no-purchase alternative. Then, the purchase probability of product j given offer tuple $\boldsymbol{x}$ is described by $p_{l\boldsymbol{x}}(j)$, where again $p_{l\boldsymbol{x}}(0)$ denotes the probability of no-purchase. In MNL, the concrete probabilities are derived by the following formulas

$$p_{l\boldsymbol{x}}(j) = \frac{u_{lj}x_j}{u_{l0} + \sum_{p\in[n]} u_{lp}x_p} \ \forall j \in [n] \quad \text{(probability of purchasing product } j\text{)} \tag{2.7}$$

$$p_{l\boldsymbol{x}}(0) = 1 - \sum_{j\in[n]} p_{l\boldsymbol{x}}(j) \quad \text{(probability of no-purchase)} \tag{2.8}$$

finer details

We want to point out a couple of smaller things. Note that one often finds the term *consideration set* C_l defined in this setting. It describes the set of all products being considered for purchase by customers of segment l, but this distinction is mathematically irrelevant: Utilities are assigned by the logic $u_{lj} > 0$ if customers of segment l might purchase product j (the higher, the more interested) and $u_{lj} = 0$ if customer is not interested in product. Thus, the consideration set is simply given by $C_l = j \in [n] : u_{lj} > 0$ and we could reduce the summations in Equation (2.7) and in Equation (2.8) to go just over C_l as the other summands equal 0 either way. The preference weights of one customer segment can be summarized by the vector $\boldsymbol{u}_l = (u_{l1}, \dots, u_{ln})^\intercal$ and no-purchase preference of u_{l0}. Furthermore, this setting is over-parametrized as described in the following: There are $L + 1$ free parameters, but still one degree of freedom as the two different utility vectors $\hat{\boldsymbol{u}} = 2\boldsymbol{u}$ result in the same probability measures: $\hat{p}_{l\boldsymbol{x}}(j) = \frac{\hat{u}_{lj}x_j}{\hat{u}_{l0}+\sum_{p\in[n]}\hat{u}_{lp}x_p} = \frac{2u_{lj}x_j}{2u_{l0}+\sum_{p\in[n]}2u_{lp}x_p} = \frac{2u_{lj}x_j}{2\left(u_{l0}+\sum_{p\in[n]}u_{lp}x_p\right)} = \frac{u_{lj}x_j}{u_{l0}+\sum_{p\in[n]}u_{lp}x_p} = p_{l\boldsymbol{x}}(j)$. Thus, we can normalize

utilities by e. g. setting $u_{l0} = 1$ (which is possible, if $u_{l0} > 0$ is assumed).[3]

Together with the uncertainty of which customer segment arrives (if any), we end up at a purchase probability for product j given offerset $\boldsymbol{x}$ of $p_{\boldsymbol{x}}(j)$ and a no-purchase probability of $p_{\boldsymbol{x}}(0)$ as presented in the following. Thus, $p_{\boldsymbol{x}}(j)$ can be interpreted as the deterministic quantity of product j being sold, if set $\boldsymbol{x}$ is offered.

$$p_{\boldsymbol{x}}(j) = \sum_{l \in [L]} \lambda_l p_{l\boldsymbol{x}}(j) \tag{2.9}$$

$$p_{\boldsymbol{x}}(0) = 1 - \sum_{j \in [n]} p_{\boldsymbol{x}}(j) \tag{2.10}$$

2.2 Solution Methods

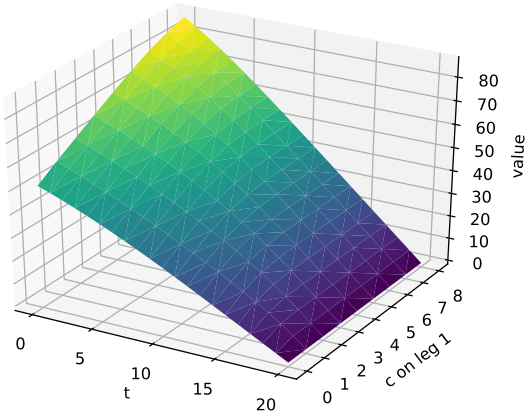
2.2.1 Exact solution

The first solution approach that might come to ones mind is to maximize the expected value of all revenues to gain given time period t and capacity $\boldsymbol{c}$ directly. So the the value function $V(t, \boldsymbol{c})$ is computed recursively starting at the final period T and all possible capacities and then working backwards in time until period 1. We name this approach *Exact Solution* (ES). The resulting value function can be plotted in a three dimensional plot as done in Figure 2.3.

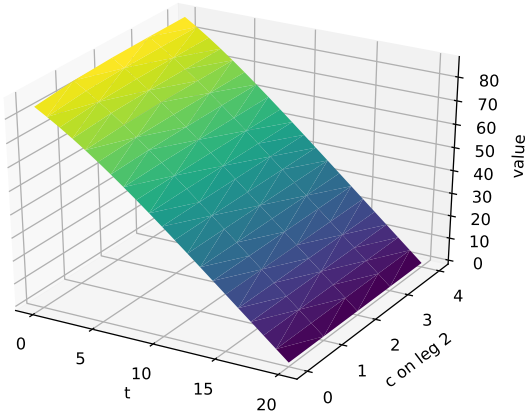
Two issues make ES difficult in practice as pointed out e. g. in Koch (2017). First, the decision problem inherent at each state (time period t together with capacity $\boldsymbol{c}$) is an assortment optimization problem over 2^n possible offer sets. So our small example given by Figure 2.1b leads to $2^6 = 64$ offer sets. Second, for each time period t, the value function Equation (2.4) has to be computed for all possible capacities, which becomes computationally cumbersome due to multi-dimensionality (m resources). In general, $\prod_{h \in [m]}(c_h^0 + 1)$ instances have to be solved[4]. Already our small example given by Figure 2.5c leads to a total of $9 \cdot 5 \cdot 5 \cdot 9 = 2.025$ instances.

[3] To give a concrete example: In our example as presented in Figure 2.5d, and therein for customer segment 1, we could change the preference vector to $(2, 4)$ and the no-purchase preference to 1. Ceteris paribus, this would leave the purchase probabilities of customer segment 1 unchanged.

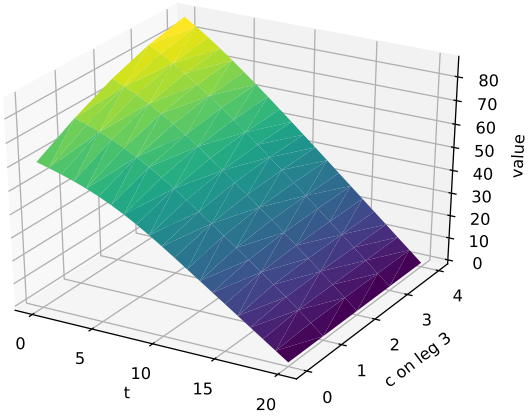
[4] The capacity of each resources can take values in $\{0, 1, \ldots, c_h^0\}$



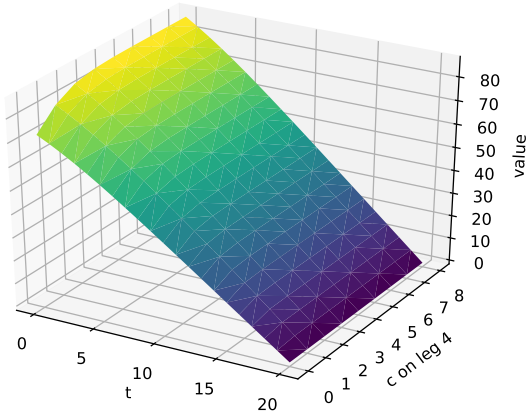
(a) Varying capacity on leg 1.



(b) Varying capacity on leg 2.



(c) Varying capacity on leg 3.



(d) Varying capacity on leg 4.

Figure 2.3: Value function of working example with time $t \in [20]$ on x-axis, the capacity as stated on the y-axis (remaining capacities set to their maximal value) and the value depicted on z-axis.

2.2.2 Choice based linear programming

CDLP Model and direct implementation

Because of the challenges for the exact solution as stated above, approximations are needed to solve Equation (2.4). A popular approach is to approximate the stochastic quantities (purchase probabilities) by deterministic values such as their expected value and to allow capacity and demand to be continuous. This approach is named *Choice based linear programming* (CDLP) and is used by e. g. Gallego et al. (2004), Liu and van Ryzin (2008) or Bront et al. (2009).

Taking an eagle-eye perspective, the company has to decide which sets to offer in every single time period. But as purchase probabilities don't change over time, the specific time period when to offer an individual set is indistinguishable, we can aggregate over time. We introduce a few more notation to get a hand on this and keep it close to what is established in the literature. As already stated in Section 2.1.3, $p_{\boldsymbol{x}}(j)$ can be interpreted as the deterministic quantity of product j being sold in one time period. Let $R(\mathbf{x})$ represent the expected (thus deterministic) revenue given offer set $\boldsymbol{x}$ (again per time period), and let $\boldsymbol{Q}(\boldsymbol{x}) = (Q_1(\boldsymbol{x}), \ldots, Q_m(\boldsymbol{x}))^\intercal$ denote the expected capacity being used (per time period), i. e.

$$R(\boldsymbol{x}) = \sum_{j \in [n]} r_j p_{\boldsymbol{x}}(j) \tag{2.11}$$

$$Q_h(\boldsymbol{x}) = \sum_{j \in [n]} a_{hj} p_{\boldsymbol{x}}(j) \ \forall h \in [m] \ . \tag{2.12}$$

We can denote the total time[5] for which set $\boldsymbol{x}$ is offered by $t(\boldsymbol{x})$ and the set of all possible offersets as $N = \{0,1\}^n$. Thus, we can formulate the choice-based linear program $V^{CDLP}(T, \boldsymbol{c})$[6]. Here, the optimization problem is solved by varying the decision variables $t(\boldsymbol{x})$.

$$V^{CDLP}(T, \boldsymbol{c}): \qquad \max \sum_{\boldsymbol{x} \in N} R(\boldsymbol{x}) t(\boldsymbol{x}) \tag{2.13}$$

$$\text{s.t.} \sum_{\boldsymbol{x} \in N} Q_h(\boldsymbol{x}) t(\boldsymbol{x}) \leq c_h \ \forall h \in [m] \tag{2.14}$$

$$\sum_{\boldsymbol{x} \in N} t(\boldsymbol{x}) \leq T \ , \tag{2.15}$$

$$t(\boldsymbol{x}) \geq 0 \ \forall \boldsymbol{x} \in N \ . \tag{2.16}$$

We want to mention some characteristics of this optimization problem. By allowing $t(\boldsymbol{x})$ to be continuous[7], we arrive at a classical linear program (LP), for which efficient solution

[5] Total time meaning number of time periods.

[6] Note that in our formulation, $p_{\boldsymbol{x}}(j)$ already includes the parameter for the arrival rate λ and thus, λ doesn't appear directly in our formulation of the CDLP in contrast to the fomulation in Bront et al. (2009).

[7] e. g., $t(\boldsymbol{x}) = 1.2$ corresponds to offerset $\boldsymbol{x}$ being offered for one whole time period and 20% of another time period.

methods exist (such as Simplex) and are implemented (e. g. in Gurobi). Even though the number of variables is exponential (2^n as $N = \{0,1\}^n$), methods such as column-generation techniques can be used to solve this LP efficiently as Gallego et al. (2004) and Liu and van Ryzin (2008) show[8]. Also note that the optimization problem $V^{CDLP}(T, \boldsymbol{c})$ has $m+1$ constraints (m in Equation (2.14) and 1 in Equation (2.15)). By duality theory and especially Complementary Slackness (compare e. g. to Domschke et al. (2015)), at most $m+1$ basic variables will end up being non-zero. Hence, even though the number of variables being exponentially large (2^n), only a linear number ($m+1$) will make up the solution, which again motivates to use column-generation techniques.

The dual of the CDLP is given by

$$
\begin{aligned}
VD^{CDLP}(T, \boldsymbol{c}): \quad & \min \boldsymbol{\pi}^\intercal \boldsymbol{c} + \sigma T && (2.17)\\
& \text{s.t.} \boldsymbol{\pi}^\intercal \boldsymbol{Q}(\boldsymbol{x}) + \sigma \geq R(\boldsymbol{x}) \ \forall \boldsymbol{x} \in N && (2.18)\\
& \boldsymbol{\pi} \geq 0 && (2.19)\\
& \sigma \geq 0 \ , && (2.20)
\end{aligned}
$$

where $\boldsymbol{\pi} \in \mathbb{R}^m$ is the vector of dual prices corresponding to the resource constraints Equation (2.14) and σ is the dual price corresponding to the time constraint Equation (2.15). Again with theory of duality in mind, the optimal value of π_h estimates the marginal value of resource h and the optimal value of $\sigma \in \mathbb{R}$ provides a marginal value of time (i. e. one additional time period would lead to an expected increase in revenue of σ units).

In our small example, there are $256 = 2^8$ offersets and the optimal solution is presented in Figure 2.4.

2.2.3 Solving CDLP by Column Generation

The huge problem size, note again the 2^n primal variables in Equation (2.13) might cause problems, but can be taken care of by column generation techniques as suggested by Gallego et al. (2004) and pointed out by Bront et al. (2009). We first sketch the algorithm, then elaborate on its complexity and finally present approaches for solving the problem.

[8] Note that Gurobi is high-level and thus programmed in such way so that it realizes which method shall be used to solve a given LP efficiently.

CDLP Column Generation Algorithm

The idea of handling the exponential problem size lies in starting with a small number (e. g. one) of initial offer tuples and incrementally add promising offer tuples in a clever way. We start with one offer tuple[9], i. e. only one primal variable and solve a reduced linear program using only this column. After checking for the existence of any other offersets with positive reduced cost relative to the current dual prices of the reduced problem, we add a corresponding positive reduced cost primal variable and re-optimize the LP. If no offerset with positive reduced cost exists, the current solution of the problem is optimal.

The reduced primal CDLP problem at step k, i. e. with k offersets given by the set $\mathcal{N} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$, is given by

$$
\begin{aligned}
V^{CDLP-R}(T, \boldsymbol{c}): \quad & \max \sum_{\boldsymbol{x} \in \mathcal{N}} R(\boldsymbol{x}) t(\boldsymbol{x}) && (2.21)\\
\text{s.t.} \quad & \sum_{\boldsymbol{x} \in \mathcal{N}} Q_h(\boldsymbol{x}) t(\boldsymbol{x}) \leq c_h \ \forall h \in [m] && (2.22)\\
& \sum_{\boldsymbol{x} \in \mathcal{N}} t(\boldsymbol{x}) \leq T \ , && (2.23)\\
& t(\boldsymbol{x}) \geq 0 \ \forall \boldsymbol{x} \in \mathcal{N} \ . && (2.24)
\end{aligned}
$$

Again, with $\boldsymbol{\pi} \in \mathbb{R}^m$, resp. $\sigma \in \mathbb{R}$ denoting the dual prices of resources, resp. time, the corresponding dual problem is given by

$$
\begin{aligned}
VD^{CDLP-R}(T, \boldsymbol{c}): \quad & \min \boldsymbol{\pi}^\intercal \boldsymbol{c} + \sigma T && (2.25)\\
& \text{s.t.} \boldsymbol{\pi}^\intercal \boldsymbol{Q}(\boldsymbol{x}) + \sigma \geq R(\boldsymbol{x}) \ \forall \boldsymbol{x} \in \mathcal{N} && (2.26)\\
& \boldsymbol{\pi} \geq 0 && (2.27)\\
& \sigma \geq 0 \ . && (2.28)
\end{aligned}
$$

Let us analyse $V^{CDLP-R}(T, \boldsymbol{c})$ in more depth to explore the ideas behind column generation algorithms. Equation (2.25) shall be minimized by altering the (dual) variables $\boldsymbol{\pi}$ and σ. These variables are bounded from below by 0 and have to satisfy Equation (2.26). At this point, we introduce the reduced costs of an offerset $\boldsymbol{x}$ as $rc(\boldsymbol{x}) := R(\boldsymbol{x}) - \boldsymbol{\pi}^\intercal \boldsymbol{Q}(\boldsymbol{x}) - \sigma$

[9] In alignment with Bront et al. (2009), this offerset comprises the first product of interest of each customer segment.

and realize that Equation (2.26) is equivalent to $rc(\boldsymbol{x}) \leq 0$, i. e. all offersets in $\mathcal{N}$ are ensured to have negative reduced costs (the dual variables are chosen in such manner). Remember $N = 0, 1^n$, thus we skipped all offersets $\boldsymbol{x} \in N \backslash \mathcal{N}$. If an offerset $\hat{\boldsymbol{x}}$ has negative reduced costs, adding $\hat{\boldsymbol{x}}$ to $\mathcal{N}$ results in one additional constraint in Equation (2.26), which is inactive, i. e. satisfied (we added variable $\hat{\boldsymbol{x}}$ with negative reduced costs). Thus, the objective value wouldn't change as the additional constraint doesn't put a restriction on the optimal dual variables $\boldsymbol{\pi}$ and σ. In the contrary, when considering an offerset $\check{\boldsymbol{x}}$ with positive reduced costs, adding the corresponding constraint in the dual problem results in a violation of Equation (2.26). Thus, the corresponding constraint is active and $\boldsymbol{\pi}$ and σ have to be adjusted to become admissible again.

Having this idea in mind motivates the column generation subproblem:[10]

$$\max_{\boldsymbol{x} \in N} rc(\boldsymbol{x}) = \max_{\boldsymbol{x} \in N} \{R(\boldsymbol{x}) - \boldsymbol{\pi}^\intercal \boldsymbol{Q}(\boldsymbol{x})\} - \sigma \tag{2.29}$$

If the optimal solution to Equation (2.29) is positive, we augment $\mathcal{N}$ with the corresponding $\hat{\boldsymbol{x}} = \arg\max_{\boldsymbol{x} \in N} rc(\boldsymbol{x})$ (note once more that $rc(\hat{\boldsymbol{x}}) > 0$ and thus $\hat{\boldsymbol{x}}$ is not part of the old $\mathcal{N}$).

If the optimal solution to Equation (2.29) is non-positive, the current solution is already optimal. Let's elaborate on this. $V^{CDLP}(T, \boldsymbol{c})$ differs from $V^{CDLP-R}(T, \boldsymbol{c})$ only in the considered set of offersets, all N versus the reduced $\mathcal{N}$. We augmented $\mathcal{N}$ to a point, when the column generation subproblem *Equation* (2.29) results in a non-positive solution. Now, Equation (2.26) is satisfied $\forall \boldsymbol{x} \in \mathcal{N}$ (by construction), but also $\forall \boldsymbol{x} \in N$ (compare to the optimal solution of the column generation subproblem). In summary, the optimal value of $VD^{CDLP-R}(T, \boldsymbol{c})$ equals the optimal value of $VD^{CDLP}(T, \boldsymbol{c})$. Furthermore, $VD^{CDLP}(T, \boldsymbol{c})$ equals $V^{CDLP}(T, \boldsymbol{c})$ as strong duality holds because we are dealing with a linear optimization problem and have found a feasible solution for the dual, which is a sufficient condition for strong duality as proven in Theorem 6.1.7*b*) of Gritzmann (2013).

Note that, up to now, we have not considered a particular customer choice model. Let us continue with introducing the MNL choice model also in this (CDLP) context, in order to analyse it's complexity in the following. With MNL as customer choice model, Equation (2.29) can be expressed as (all equivalent)

[10] Note that in comparison to Bront et al. (2009), the potential offersets to consider can be limited to $N \backslash \mathcal{N}$ as $\boldsymbol{\pi}$ and σ are chosen in such manner that $rc(\boldsymbol{x}) \leq 0$ is ensured $\forall \boldsymbol{x} \in \mathcal{N}$.

$$\max_{\boldsymbol{x}\in\{0,1\}^n} \left\{ \sum_{j\in[n]} r_j p_{\boldsymbol{x}}(j) - \sum_{h\in[m]} a_{hj}\pi_h p_{\boldsymbol{x}}(j) \right\} - \sigma \tag{2.30}$$

$$\max_{\boldsymbol{x}\in\{0,1\}^n} \left\{ \sum_{j\in[n]} \left(r_j - A_j^\intercal \boldsymbol{\pi}\right) p_{\boldsymbol{x}}(j) \right\} - \sigma \tag{2.31}$$

$$\max_{\boldsymbol{x}\in\{0,1\}^n} \left\{ \sum_{l\in[L]} \lambda_l \frac{\sum_{j\in[n]} \left(r_j - A_j^\intercal \boldsymbol{\pi}\right) u_{lj} x_j}{\sum_{j\in[n]} u_{lj} x_j + u_{l0}} \right\} - \sigma \tag{2.32}$$

$$\max_{\boldsymbol{x}\in\{0,1\}^n} \left\{ \sum_{j\in[n]} \left(r_j - A_j^\intercal \boldsymbol{\pi}\right) x_j \left(\sum_{l\in[L]} \frac{\lambda_l u_{lj}}{\sum_{j\in[n]} u_{lj} x_j + u_{l0}} \right) \right\} - \sigma \tag{2.33}$$

Note that the assumptions $u_{lj} \in \mathbb{R}_0^+ \; \forall j \in [n]$, resp. $u_{l0} \in \mathbb{R}^+$ ensure the optimization problem to always be well defined.

Complexity of Column Generation

As noted in Bront et al. (2009), Equation (2.33) is in general not separable in the variables $x_j, j \in [n]$ as multiple customer segments (e. g. l_1 and l_2) might be interested in the same product j and thus, x_j appears in the denominator belonging to l_1 and in the one belonging to l_2. Our column generation problem is part of the general group of *hypberbolic (or fractional) 0-1 programming problems*, which were studied by e. g. Hansen et al. (1991) or Borrero et al. (2017) and the most general version is given by

$$\max_{\boldsymbol{x}\in\{0,1\}^n} \sum_{l\in[L]} \frac{a_{l0} + \sum_{j\in[n]} a_{lj} x_j}{b_{l0} + \sum_{j\in[n]} b_{lj} x_j} \;, \tag{2.34}$$

with no restrictions on a_{lj}, b_{lj} and $x_j \in \{0,1\} \forall j \in [n]$. Prokopyev et al. (2005b) have proven that Equation (2.34) is NP-hard by providing a polynomial reduction from the weighted 2SAT problem.

Bront et al. (2009) point out that due to the tight linkage of variables in our column generation subproblem, the general Equation (2.34) cannot be easily reduced to the specific Equation (2.33). But they reduce the *minimum vertex cover problem* to our problem Equation (2.33). And as Garey and Johnson (2009) proofs the *minimum vertex cover*

problem to be NP-hard[11], our column generation problem is NP-hard as well.

Approaches for solving Column Generation

MIP Formulation. We follow Bront et al. (2009) to formulate the column generation problem Equation (2.29) as a mixed integer problem (MIP). To get rid of the fraction, we introduce the new variable

$$y_l = \frac{1}{\sum_{j\in[n]} u_{lj}x_j + u_{l0}} \ , \tag{2.35}$$

and enforce this via the (non-fractional) constraint in Equation (2.41). By also changing the order of summation[12], using the distributive law and ignoring the constant σ (we check if the optimal result is $> \sigma$ in the end), we rewrite the column generation problem Equation (2.33) as[13]

$$\max_{\boldsymbol{x},\boldsymbol{y}} \sum_{l\in[L]} \sum_{j\in[n]} (r_j - A_j^\intercal \boldsymbol{\pi}) x_j \lambda_l u_{lj} y_l \tag{2.36}$$

$$\text{s.t. } y_l u_{l0} + \sum_{j\in[n]} u_{lj} x_j y_l = 1 \ \forall l \in [L] \tag{2.37}$$

$$x_j \in \{0,1\} \ \forall j \in [n] \tag{2.38}$$

$$y_l \geq 0 \ \forall l \in [L] \ . \tag{2.39}$$

It remains to linearise the nonlinear term (product) $x_j y_l$. We first introduce the concept of linearizing xy with $x \in \{0,1\}$ and $y \geq 0$ following Klein (1718). We introduce the auxiliary variable $z \geq 0$. Obviously, we want $z = 0$ if $x = 0$. This leads to the constraint $z \leq xy$. And we want $z = y$ if $x = 1$. This can be enforced by putting the constraints $z \leq y$[14] and $z \geq y - M(1-x)$ with M large enough (this idea of introducing a large

[11] Stated in [GT1] Vertex Cover under *A1.1 Covering and Partitioning* on page 190 and referred to Karp (2010), who proves NP-hardness by a transformation from 3SAT, the well established example for a problem of the NP-class.

[12] Changing the order of summation is allowed since the sum is finite.

[13] Note that Bront et al. (2009) used sloppy notation by claiming $j \in N$, as she introduced N to be a constant (the total number of products). Our notation is precise.

[14] Note that this constraint is already included in the previous one $z \leq yx$, thus can be excluded here. This thought can be used to enhance Klein (1718) by combining constraints (1.20) and (1.20) to $\lambda_{ij} \leq p_j x_{ij}$. Note that constraint (1.20) does not really come from enforcing the linearization, but

constant to switch off a constraint if need be is generally referred to as *big M*). Note that the other constraints are trivially fulfilled if $x = 1$, resp. $x = 0$. We apply these ideas and introduce the auxiliary variable $z_{lj} = x_j y_l$ with the corresponding constraints and obtain the MIP formulation

$$\max_{\boldsymbol{x},\boldsymbol{y}} \sum_{l\in[L]} \sum_{j\in[n]} (r_j - A_j^\intercal \boldsymbol{\pi})\lambda_l u_{lj} z_{lj} \tag{2.40}$$

$$\text{s.t. } y_l u_{l0} + \sum_{j\in[n]} u_{lj} z_{lj} = 1 \ \forall l \in [L] \tag{2.41}$$

$$y_l - z_{lj} \leq M(1 - x_j) \ \forall l \in [L] \forall j \in [n] \tag{2.42}$$

$$z_{lj} \leq y_l x_j \ \forall l \in [L] \forall j \in [n] \tag{2.43}$$

$$x_j \in \{0, 1\} \ \forall j \in [n] \tag{2.44}$$

$$y_l \geq 0 \ \forall l \in [L] \tag{2.45}$$

$$z_{lj} \geq 0 \ \forall l \in [L] \forall j \in [n] \ , \tag{2.46}$$

with $M \geq 1/\min\{u_{l0} : l \in [L]\}$[15]. M is large enough, because Equation (2.42) is satisfied if activated ($x_j = 0$) as z_{lj} is then forced to zero by Equation (2.43) and y_l potentially set to $1/u_{l0}$ by Equation (2.41).

Having such a MIP formulation allows for the usage of standard MIP solvers like Gurobi. But as the complexity result of Bront et al. (2009) states, the column generation problem is still NP hard. Thus, we present a polynomial time heuristic.

Greedy Heuristic

To identify the most promising offerset to add to the current set of offersets $\mathcal{N}$, we use a greedy, constructive heuristic. It was suggested by Bront et al. (2009), with an underlying algorithm originally proposed by Prokopyev et al. (2005a) to solve a general fractional programming problem. We adopt notation heavily to present a more concise and mathematical version. The algorithm makes straightforward use of the most promising offersets. Note that, via the current offersets to consider $\mathcal{N}$, we arrived at values of the dual variable $\boldsymbol{\pi}$ (and σ, but this is not needed).

from the participation constraint (1.14).

[15] In comparison to Bront et al. (2009), we adopt the minimum as not all products might be in the consideration set, thus $\underline{v}$ might be 0. Our denominator will always be nonzero, as all u_{l0} are assumed to be > 0, and will be appropriate as potentially adding some positive u_{lj} just increases the denominator, thus decreases M.

The heuristic itself is given in Algorithm 1[16] and explained in the following. The algorithm starts in Line 1 with defining the value of an offerset candidate X. This really just rewrites the column generation subproblem Equation (2.33) by directly summing solely over the products in the offerset candidate instead of summing over all products and just keeping those for which $x_j = 1$. Line 2 initializes the set final set S, which will comprise the offerset in the end, and the set S', which contains promising products, i. e. such products with positive reduced costs. In Line 3, the most promising product j^* is selected. Within the loop, sets S and S' are updated (Line 5) and the next suitable product j^* is determined. Whenever including this product j^* in the offerset S makes sense, i. e. raising the value of the offerset, the loop iterates again. When the value function doesn't increase further, the optimal offerset S is returned.

1: $\texttt{Value}(X) \coloneqq \sum_{l=1}^{L} \lambda_l \frac{\sum_{i \in X}(r_i - A_i^\intercal \pi) u_{li}}{\sum_{i \in X} u_{li} + u_{l0}}$
2: $S \coloneqq \emptyset, \quad S' \coloneqq \left\{ j \in [n] : r_j - A_j^\intercal \pi > 0 \right\}$
3: $j^* \coloneqq \arg\max_{j \in S'} \texttt{Value}(\{j\})$
4: **repeat**
5: $\quad S \coloneqq S \cup \{j^*\}, \quad S' \coloneqq S' \backslash \{j^*\}$
6: $\quad j^* \coloneqq \arg\max_{j \in S'} \texttt{Value}(S \cup \{j\})$
7: **until** $\texttt{Value}(S \cup \{j^*\}) \leq \texttt{Value}(S)$
8: **return** S, e. g. as offerset $\boldsymbol{x}$ given by $x_j = \mathbb{1}_{\{j \in S\}}, \ j \in [n]$

Algorithm 1: Greedy Heuristic

This heuristic has worst-case complexity of $O(n^2 L)$ as the repeat loop might run for all n products and in Line 6 the calculation of the arg max of `Value` is done $|S'|$ times with a summation over all L customer segments and the remaining, $n - |S'|$ products. Note that in practice, there are much less customer segments than products, i. e. $L << n$, leading to less then cubic complexity in n.

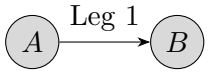
We want to present an adopted version[17] of the example in Bront et al. (2009) to show that Algorithm 1 is indeed a heuristic, i. e. stops without finding the optimal solution. We also use this example as first validation of our code. Consider the column generation subproblem Equation (2.33) with $n = 3, L = 3$, other parameters as specified in Figure 2.5 and $\boldsymbol{\pi} = (0, 0, 0)^\intercal$. Applying our implemented functions result in Figure 2.6, which proofs

[16] Note that our first j^*, found in Line 3, is in consistence with the other j^*'s found in Line 6, i. e. including the arrival probabilities λ_l. This is not the case in Bront et al. (2009), compare step 3 with step 4.a.

[17] We adjust parameters $\lambda_l = 1/3$ to sum up to 1. Note, this is necessary due to our assumption of at most one customer arriving at a given point in time. To reproduce the optimal values stated in Bront et al. (2009), we multiply the objective value by 3 due to the scalability of Poisson processes as stated in .

that the greedy heuristic was unable to find the optimum as it stopped at the optimal value of 16.66 instead of arriving at the truly optimal 17.83.

(a) Airline network.



(b) Products.

Product	Origin-destination	Fare
1	$A \to B$	100
2	$A \to B$	19
3	$A \to B$	19

(c) Resources.

Leg	Capacity
1	∞

(d) Customers.

Seg.	λ_l	Consideration tuple	Preference vector	No purchase preference	Description
1	1/3	$(1, 2, 3)$	$(1, 1, 1)$	1	Flexible, (A→B)
2	1/3	(2)	(1)	1	Just 2, (A→B)
3	1/3	(3)	(1)	1	Just 3, (A→B)

Figure 2.5: Small example for illustration of sub-optimality of CDLP Greedy Heuristic with $T = 1$ time period.

```
MIP results in: optimal value =  17.83    optimal tuple =  (1, 1, 1)
GH results in:  optimal value =  16.66    optimal tuple =  (1, 0, 0)
```

Figure 2.6: Results for small example. MIP for "CDLP with MIP formualtion" vs. GH for "CDLP with greedy heuristic".

Our final implementation of the CDLP by column generation consists of: First, we use the greedy heuristic to identify a promising offerset. If this method doesn't succeed, we use the exact MIP procedure. If no new offerset is identified to enter the base, we have found the optimal solution of the CDLP.

Applying this version of CDLP by column generation with the greedy heuristic to our running Example, we arrive at the optimal solution as presented in Figure 2.7. Note that the same solution is found as by using CDLP in its MIP formulation, compare to Figure 2.4. But the column generation version with the greedy heuristic needs just 3 columns in comparison to 256 used by the MIP version.

```
Optimal objective value:
 91.95238095238096

Found by using  3  columns.

Optimal solution:
Tuple (1, 0, 0, 1, 1, 0)  is offered for a total time of  20.0


Product  1 is offered during  20.0000  time periods.
Product  2 is offered during   0.0000  time periods.
Product  3 is offered during   0.0000  time periods.
Product  4 is offered during  20.0000  time periods.
Product  5 is offered during  20.0000  time periods.
Product  6 is offered during   0.0000  time periods.
```

Figure 2.7: Optimal solution for CDLP by column generation for working example.

2.3 Approximate Dynamic Programming

Secondly

We now move on to a new solution approach, namely approximate dynamic programming. The method is first motivated and put in context of the thesis. Second, a short classification of different methods belonging to this class is presented, before the method applied in our setting is discussed in detail.

ADP builds upon Equation (2.6). The real challenge is to determine the opportunity costs per product j, i. e. $\Delta_j V(t+1, \boldsymbol{c})$. ADP approximates these opportunity costs additively using bid prices for the resources h, $\pi_h(t, c_h)$, if those are needed to produce product j, i. e. $\Delta_j V(t+1, \boldsymbol{c}) = \sum_{h\in[m]} a_{hj}\pi_h(t+1, c_h)$. Thus, the policy is given as the solution of

$$ {}^{\pi}\boldsymbol{x}^t = \underset{\boldsymbol{x}^t\in\{0,1\}^n}{\arg\max} \left\{ \sum_{j\in[n]} p_j(\boldsymbol{x}^t) \left(r_j - \sum_{h\in[m]} a_{hj}\pi_h(t+1, c_h) \right) \right\} , \tag{2.47} $$

and is therefore completely described by the bid prices $\pi_h(t, c_h)$ $\forall t \in [T]$. Note that $\boldsymbol{x}^t$ represents the set offered during time period t, i. e. the decision is made at time point $t-1$. $\Delta_j V(t+1, \boldsymbol{c})$ represents the difference in revenue to go when looking at selling product j and considering time periods $t+1, \ldots, T$ and capacities $\boldsymbol{c}$. Note that when considering the

very last time period T, opportunity costs $\Delta_j V(t+1, \boldsymbol{c})$ are zero $\forall j \in [n], \forall \boldsymbol{c} \geq 0$.

We now present a brief overview how to classify simulation based ADP methods, which the textbooks Bertsekas (2005) and Powell (2011) cover in more detail. The methods are named *approximate*, as they are based on approximating the value function. Various sample paths are generated and used in basically two different manners to iteratively improve the approximation of the value function: either by *approximate value iteration* (AVI) or by *approximate policy iteration* (API). Each iteration of AVI comprises a single sample path. It evaluates the current policy on this particular sample path and updates parameters for the value function approximation after each period of the sample path. In contrary, each iteration of API comprises multiple sample paths. It evaluates the current policy on a set of sample paths and uses the combined information to update and improve the value function approximation (and thus the policy). Thus, one iteration of API is certainly costlier and usually API requires less iterations to obtain good policies. One obvious prerequisite for this is a large enough set of sample paths. We refer to Powell (2011) for more details.

A suitable method to be used for our setting was proposed by Koch (2017) and belongs to the class of simulation based ADP. More precisely, it comprises a value function approximation algorithm and uses policy iteration. It approximates the value function $V(t, \boldsymbol{c})$ by computationally simple functions (linear or piecewise linear), makes use of an offline phase to determine bid prices $\pi_h(t, c_h)$ $\forall t \in [T]$ and applies Equation (2.47) in an online phase to compute the offerset.

2.3.1 Approximate Policy Iteration

Approximate Policy Iteration is used in the offline phase and aims at determining bid prices for each time period $\pi_h(t, c_h)$ $\forall t \in [T]$, which fully determine the policy used in the online phase. The complete algorithm is given by Algorithm 2 and comprises several steps. We first give an overview of the concept and then explain the details step by step.

API value function approximation

We will introduce the concept of the method in the following and created Table 2.2 for reference, as there are quite some variables involved in API.

Symbol	Domain	Meaning
θ_t	$\mathbb{R}$	optimization variable (offset) for time period t
Θ	$\mathbb{R}^\intercal$	optimization variable comprising all θ_t
$\boldsymbol{\pi}_t$	$\mathbb{R}^m$	optimization variable (bid price for each resource) for time period t
$\boldsymbol{\Pi}$	$\mathbb{R}^{m \times T}$	optimization variable (bid price for each resource) comprising all $\boldsymbol{\pi}_t$
$\hat{V}_t^i$	$\mathbb{R}$	sample revenue to go for sample i starting at and including period t
$\hat{V}$	$\mathbb{R}^{T \times I}$	all sample revenues to go comprising all $\hat{V}_t^i$
$\hat{\boldsymbol{C}}_{t-1}^i$	$\mathbb{R}^m$	available capacities for resources for sample i at end of period $t-1$
$\hat{\boldsymbol{C}}$	$\mathbb{R}^{m \times T \times I}$	all available capacities comprising all $\hat{\boldsymbol{C}}_{t-1}^i$
r_t	$\mathbb{R}$	sample revenue generated during time period t
$\boldsymbol{c}$	$\mathbb{N}^m$	available capacities for each resource at current time period
$\boldsymbol{c}^0$	$\mathbb{N}^m$	starting capacities
$\boldsymbol{x}$	$\{0,1\}^n$	offerset at current time period
ϵ_t	$[0,1]$	epsilon used at time period t

Table 2.2: API overview of parameters.

Set $\theta_t = 0$ and $\boldsymbol{\pi}_t = \mathbf{0}$ $\forall t \in [T]$
for `k = 1 to K` **do**
  Set $\hat{V}_t^i = 0$ and $\hat{\boldsymbol{C}}_{t-1}^i = 0$ $\forall t \in [T], \forall i \in [I]$
  for `i = 1 to I` **do**
    Set $\hat{r}_t = 0$ and $\hat{\boldsymbol{c}}_{t-1} = 0$ $\forall t \in [T]$
    Initialize $\boldsymbol{c} = \boldsymbol{c}^0$
    for `t = 1 to T` **do**
      $\hat{\boldsymbol{c}}_{t-1} \coloneqq \boldsymbol{c}$
      Get $\boldsymbol{\pi}(t, \boldsymbol{c})$
      Compute $\boldsymbol{x} =$ `determineOfferset`$(\boldsymbol{\pi}(t, \boldsymbol{c}), \epsilon_t)$
      Simulate a sales event $j' \in \{0, 1, \ldots, n\}$
      if $j' \in \{1, \ldots, n\}$ **then**
        $\hat{r}_t = r_{j'}$ and $\boldsymbol{c} = \boldsymbol{c} - \boldsymbol{a}_{j'}$
    Compute $\hat{V}_t^i = \sum_{\tau=t}^{T} \hat{r}_t$ $\forall t \in [T]$
    Assign $\hat{\boldsymbol{C}}_{t-1}^i = \hat{\boldsymbol{c}}_{t-1}$ $\forall t \in [T]$
  $(\Theta, \boldsymbol{\Pi}) =$ `updateParameters`$(\hat{V}, \hat{\boldsymbol{C}}, \Theta, \boldsymbol{\Pi}, k)$
return $(\Theta, \boldsymbol{\Pi})$

Algorithm 2: Approximate policy iteration. Note that t represents a time period and therefore takes values in $[T]$. As at time period t, there is c_{t-1} capacity available, we use a different index for capacity.

API aims at approximating the value function for each state. A state is characterized by current time period t and vector of current capacities $\boldsymbol{c}$. Therefore, a natural choice is to use a linear function as given by

$$V_{t,\boldsymbol{c}}(\theta_t, \boldsymbol{\pi}_t) \coloneqq \theta_t + \sum_{h \in [m]} \pi_{th} c_h \tag{2.48}$$

with

$$\theta_t \geq 0 \tag{2.49}$$

$$\max_{j=1,\dots,n} r_j \geq \pi_{th} \geq 0 \tag{2.50}$$

equation

and $\theta_{T+1} = 0$ as well as $\pi_{T+1,h} = 0 \; \forall h \in [m]$ assumed implicitly, thus leading to $V(T+1, \boldsymbol{c}) = 0$ if $\boldsymbol{c} \geq \boldsymbol{0}$. Note that we follow Koch (2017) and include expert knowledge on the problem. The non-negativity constraints in Equation (2.49) and Equation (2.50) ensure that the optimal value is non-negative and increasing in capacity. The upper bound in Equation (2.50) is also reasonable as one additional capacity should increase the optimal value less than the amount of revenue generated by selling the most expensive product. This directly leads us to the interpretation of the variables. θ_t represents the constant offset. π_{th} can be directly interpreted as bid prices of the resources, and therefore be used in Equation (2.47): $\pi_h(t, c_h) = \pi_{th}$. Note that in this setting, the bid price is independent of the current level of capacity.[18]

API overview

Let us now discuss the algorithm in depth.[19] For the first policy iteration, Line 1 sets all optimization variables to zero which results in the greedy policy of offering the set resulting in the highest expected revenue (no consideration of costs) in each time period.

Then, a total of K policy iterations follow.[20] In each policy iteration k, the current policy is evaluated first. To do so, we use the current policy to evaluate I random sample

[18] In comparison of a linear with a piecewise linear approach.

[19] We strictly use our notion of time periods in this thesis. Note that there is a one-to-one correspondence of "action happening during time period t with $t \in [T]$" and "action happening at time point t with $t \in \{0, \dots, T-1\}$". The latter is more convenient to use in the Python implementation as Python indexing starts at 0.

[20] Note that this direct approach is used in Koch (2017). While a suitable K was presumably found looking at convergence rates and then fixed, I would encourage practitioners to directly apply a convergence criterion for termination of policy iteration. Note that policy iteration is proven to converge if a discounting factor of later revenues is included. Compare Bertsekas(2005; Proposition 1.3.6 on p. 48).

paths. Line 3 sets up the dataset for revenues to go $\hat{V}_t^i$ and remaining capacities $\hat{\boldsymbol{C}}_{t-1}^i$ for all time periods $t \in [T]$ and sample paths $i \in [I]$.

For each sample path i, a storage unit for revenue $\hat{r}_t$ is created for all time periods $t \in [T]$ and all capacities $\hat{\boldsymbol{c}}_{t-1}$ (Line 5). Note, that we use a different index for capacities to account for the fact that for the remaining revenues to go starting from and including time period t, we have capacities $\boldsymbol{c}_{t-1}$ available (those being left over from time period $t-1$.). Furthermore, the capacity is set to the initial capacity in Line 6.

For each time period, the available capacity $\hat{\boldsymbol{c}}_{t-1}$ (left over at end of previous time period) is stored in Line 8, the current bid prices are calculated in **??** and used to determine the offerset $\boldsymbol{x}$ in Line 10. More details on the determination of the offerset can be found in Section 2.3.1. With this information, a sales event j' is simulated and in case a product has been sold ($j' \in [I]$), the revenue is stored and capacity adjusted accordingly. Note, the revenue thus belongs to time period t and capacity will affect time period $t+1$ (Line 11 to Line 13).

After simulating one sample path, the value function for this sample path $\hat{V}_t$ is evaluated for all time periods $t \in [T]$ in Line 14, and so are the capacities in Line 15.

Finally, Line 16 comprises the policy improvement step and uses all the information of the current policy iteration to update the optimization variables $\theta_t, \boldsymbol{\pi}_t \; \forall t$. Before expanding on how this update is executed, we want to fill the gap on how the offerset is determined in Line 10.

API determination of offersets

Our goal is to efficiently determine a reasonable set of products to offer. Thus, we use a heuristic and reduce the amount of products to consider as fast as possible. The following algorithm is based on the ideas of the greedy, largest marginal benefit heuristic for the column generation subproblem outlined in Bront et al. (2009) and presented above in Algorithm 1.

The function `determineOfferset`$(\boldsymbol{\pi}, \epsilon)$ calculates the set of products to offer depending on the current bid prices $\boldsymbol{\pi}$ via the greedy algorithm pointed out in Bront et al. (2009). As we put ourselves into a policy improvement setting and started out with the greedy strategy, the whole procedure tends to find solely other policies rather closeby, i. e. also greedy. Thus, we would leave out a big portion of the whole solution space (for each time period and each combination of capacities any subset of the n products can be offered). This is the famous *exploration vs. exploitation dilemma*. Exploitation: The current solution was found

and improved already and should be exploited even further (comparable to incremental innovation). Exploration: But also other, yet to be seen parts of the solution space should be explored (comparable to disruptive innovation). To account for this dilemma, we follow Koch (2017) and use an *epsilon-greedy strategy*. With a probability of $\epsilon/2$ either no product is offered at all or all products with positive contribution $r_j - \sum_{h\in[m]} a_{hj} \cdot \pi_h$ are offered. With a probability of $1-\epsilon$, the set determined by Algorithm 1 is offered.

Note that in the very first iteration, we do not have appropriate values for $\boldsymbol{\pi}$ and θ. Thus, we decided to set them to zero at start, corresponding to "no opportunity costs of resources".

API update of parameters

With the information of the I sample paths, all optimization variables $(\Theta, \boldsymbol{\Pi})$ are updated. Note that the old parameters are used as starting values and the current policy iteration k is also passed to potentially take care of exponential smoothing. Thus, we have $(1+h) \cdot T$ parameters (θ_t and $\boldsymbol{\pi}_t$ $\forall t \in [T]$) and TI data points, where each data point comprises the value of the value function as y-value and the current assignment of t and $\boldsymbol{\pi}_t$ as x-values.

The function `updateParameters`$(\hat{V}, \hat{\boldsymbol{C}}, \Theta, \boldsymbol{\Pi}, k)$ comprises the following least squares optimization problem, which optimizes all optimization variables, $(\theta_t, \boldsymbol{\pi}_t)_t$ with $t \in [T]$ at the same time and the optimal values $(\Theta^{update}, \boldsymbol{\Pi}^{update})$ are obtained.

$$\min \sum_{i=1}^{I} \sum_{t=1}^{T} \left(\hat{V}_t^i - V_{tc_t^i}(\theta_t, \boldsymbol{\pi}_t) \right)^2 \tag{2.51}$$

s.t. (2.52)

$$\theta_t \geq 0 \qquad \forall t \in [T] \tag{2.53}$$

$$\max_{j\in[n]} r_j \geq \pi_{th} \geq 0 \qquad \forall t \in [T], \forall h \in [m] \tag{2.54}$$

$$\theta_t \geq \theta_{t+1} \qquad \forall t \in \{1, \ldots, T-1\} \tag{2.55}$$

$$\pi_{th} \geq \pi_{t+1,h} \qquad \forall t \in \{1, \ldots, T-1\} \tag{2.56}$$

The old values θ_t^k and $\boldsymbol{\pi}_t^k$ are used to determine the optimal parameter θ_t^{update} and $\boldsymbol{\pi}_t^{update}$.

There exist multiple ~~on~~ possibilities on which values shall be used as new optimization

variables. One approach might be to just use the optimal values:

$$\theta_t^{k+1} = \theta_t^{update} \tag{2.57}$$
$$\boldsymbol{\pi}_t^{k+1} = \boldsymbol{\pi}_t^{update} \tag{2.58}$$

Another approach is to use exponential smoothing, which assigns ~~quite~~ some value to the previous iterations and updates the optimization variables less and less, i. e. its weights decrease with an increasing number of policy iterations k. Here, for the next iteration $k+1$ the values of optimization variables are calculated via:

$$\theta_t^{k+1} = \left(1 - \frac{1}{k}\right) \theta_t^k + \frac{1}{k} \theta_t^{update} \tag{2.59}$$
$$\boldsymbol{\pi}_t^{k+1} = \left(1 - \frac{1}{k}\right) \boldsymbol{\pi}_t^k + \frac{1}{k} \boldsymbol{\pi}_t^{update} \tag{2.60}$$

One can also apply the first technique (use the updates directly) during the policy iteration and finally report an average over all parameter values at the very end. Note that Lemma 1 presents the circumstances and mathematical proof on when this procedure becomes equivalent to exponential smoothing as introduced above.

$$\theta_t^{K+1} = \frac{1}{K} \sum_{k=1}^{K} \theta_t^k \tag{2.61}$$
$$\boldsymbol{\pi}_t^{K+1} = \frac{1}{K} \sum_{k=1}^{K} \boldsymbol{\pi}_t^k \tag{2.62}$$

2.3.2 Running Example

We evaluate different combinations of parameters and go with the same as Koch (2017), i. e. $K = 60$ policy iterations, $I = 800$ sample paths and a parameter for the epsilon-greedy strategy monotonously decreasing with the number of policy iteration k:

$$\epsilon_k = \begin{cases} 0.05, & k = 1, \ldots, 10 \\ 0.01, & k = 11, \ldots, 20 \\ 0, & \text{otherwise} \end{cases} \tag{2.63}$$

We prepare $I \times T$ random numbers for the customer streams, the sales stream and the

epsilon criterion. Note that these numbers are random, but remain the same over the K policy iterations. Figure 2.8 shows the distribution of a uniform random variable in the whole $I \times T$ setting. It can be clearly seen that the underlying random variable indeed follows a uniform $U(0, 1)$ distribution and can thus be used for applying the epsilon-greedy strategy with thresholds as specified above. Figure 2.9 presents the boxplots of arriving customers. For each of the I iterations, the number of arriving customers in the event horizon ($t \in [T]$) was reported and then depicted. The means can be seen to be in line with the model specification (e. g. customer 1 has arrival probability of 0.3) and the standard deviation is fine. Thus, our exemplary data represents our model specification and we can apply approximate dynamic programming.

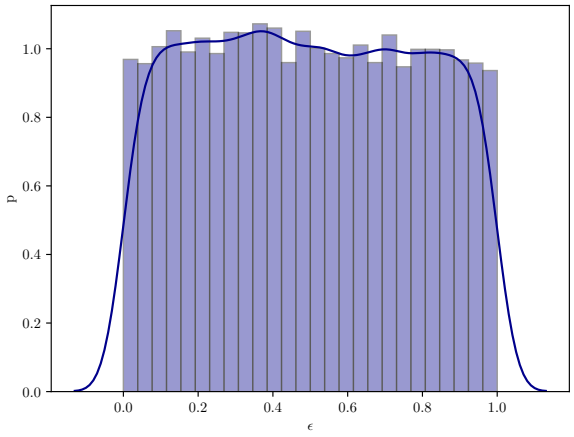


Figure 2.8: Distribution of epsilon values.

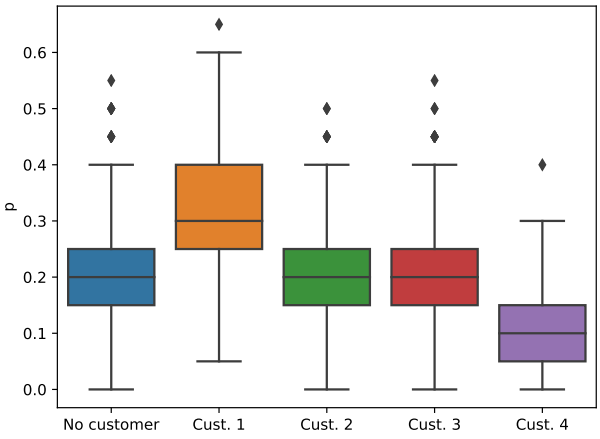


Figure 2.9: Boxplot of customers in training.

The evolution of the average realized value[21] over the I iterations in each of the $k \in \{1, \ldots, 60\}$ policy iterations is depicted in Figure 2.10. It can be seen, that the initial simplification of having opportunity costs of zero ($k = 1$), results in a pretty high overall result. This is due to the short time frame (20 time periods) in combination with plenty of available resources ($24 = 8 + 4 + 4 + 8$), so it seems reasonable to offer as many products as possible because leftovers are worthless. Roughly 15 policy iterations are needed before API comes back to a close to optimal average value at start. After some jumps of varying height, it is interesting that API starts in $k = 53$ to oscillate between the optimal values 59.265 and 69.9425. This leads to the conclusion, that no stable solution can be found. This behaviour is most probably caused by the specific interaction of which customer comes

[21] Realized value refers to value realized over the time horizon, i. e. total revenue generated due to selling of products in time periods $t \in [T]$.

when leading to varying capacities present at certain time steps, thus to different sets to be offered and ultimately purchased.

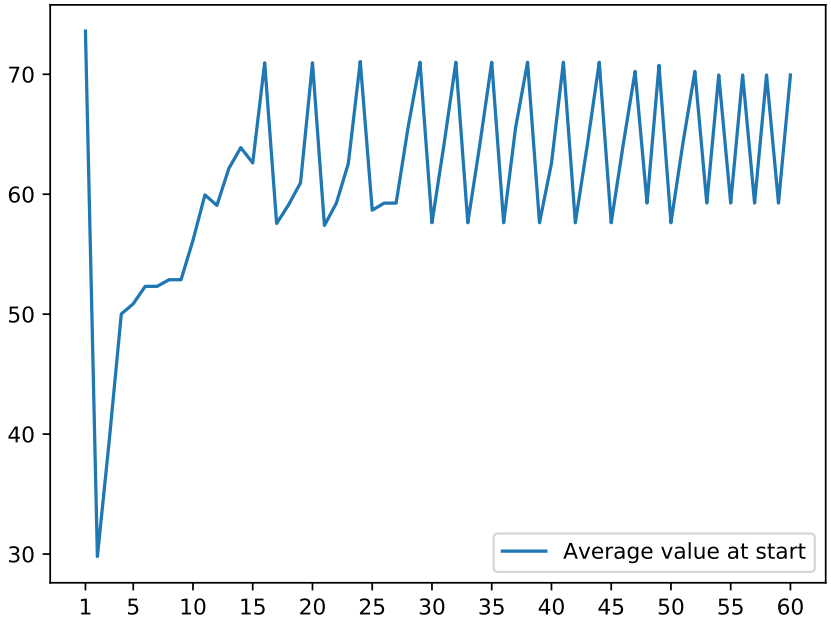
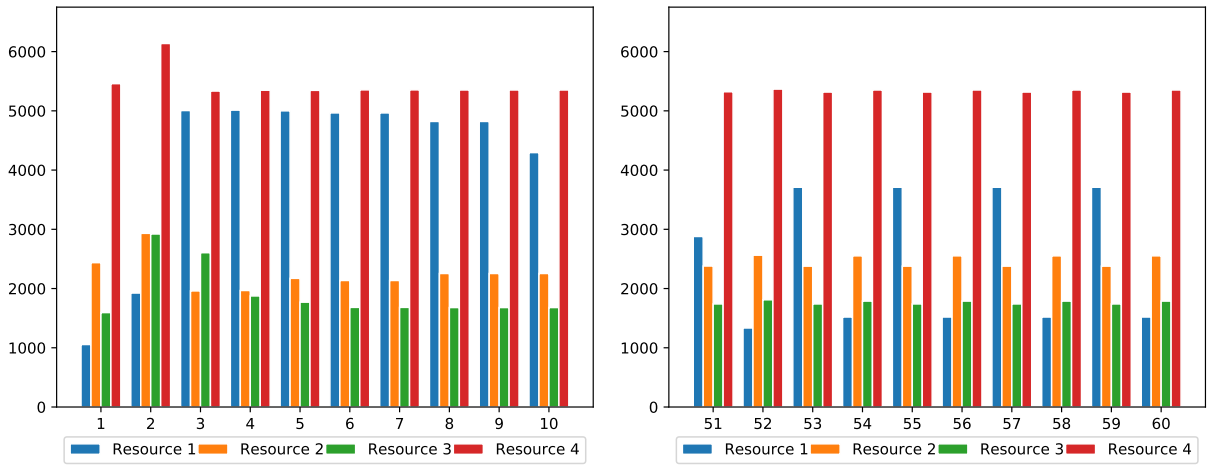


Figure 2.10: Average value at start (y-axis) for the evolution via $K = 60$ policy iterations (x-axis).

That some resources are still left over after the time horizon can be seen in Figure 2.11. The left graph depicts the remaining capacities after the selling horizon for each resource for the first 10 policy iterations (aggregated over all $I = 800$ sample paths). Remember that we start with 8 units of capacity for resources 1 and 4, and 4 units of capacity for resources 2 and 3. During the first iterations, resource 1 is used more often then resource 4, but still remains left over with quite some amount. This changes towards the latter iterations (right graph), when we see again the alternating behaviour and much less capacity of resource 1 remaining.

but a large amount
still remains left over

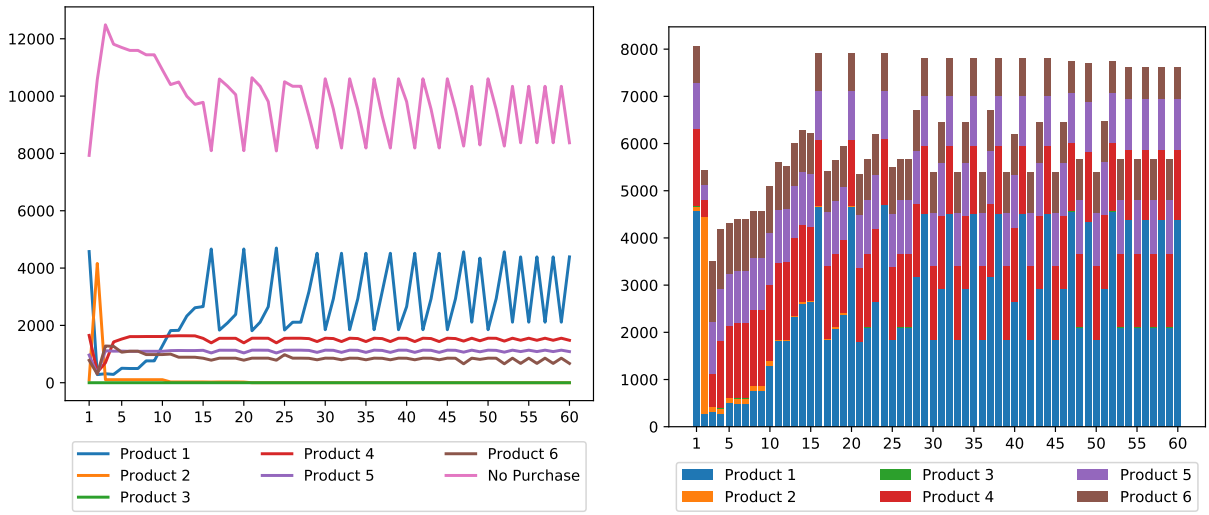


(a) $K = \{1, \ldots, 10\}$ (b) $K = \{51, \ldots, 60\}$

Figure 2.11: Remaining capacities via several policy iterations (x-axis).

This last observation goes along with the hypothesis of the oscillating solution and can also be verified by looking at Figure 2.12, which depicts the purchased products. Figure 2.12a includes the "no-purchases" explicitly, while Figure 2.12b depicts the purchased products stacked over each other, leading the total height of each cumulated bar to represent the total amount of purchased products. Note that summing up all purchase actions (including "no-purchase"), leads to a total of $16,000 (= 20 \times 800 = T \times I)$ in each polity policy ? iteration k. In combination with the value plot (Figure 2.10), we see that API is well suited for our problem. After the first iteration, which is needed to set up the parameters, product 2 is sold very often. But as this is the cheapest product, even though a lot of products are sold, the average value generated is very low (29.795). This error is recognized immediately by API and to purely offer product 2 is never done again as can be seen in Table 2.3. The policy is changed and the pretty expensive product 6 is offered most often and therefore also purchased most often ($1,281$ times, while the second most often purchased product 5 is purchased $1,116$ times). This iteration, less products then in the In this than previous iteration are sold, but a higher value is achieved. In the following iterations, the amount of purchased products increases more and more until a peak in iteration 16. The,n the alternating behaviour begins. All this is also seen in the offered products as presented in Table 2.3. Here we clearly see that the end of our optimization is reached as we oscillate back and forth from iteration 53 onwards. The algorithm is trapped in a (stable) circle.[22]

[22] As exponential smoothing is closely related to an average over all policy iterations, or at least leads to the optimal values changing less and less, the value function should converge to a unique solution



(a) With no-purchases.

(b) Solely products.

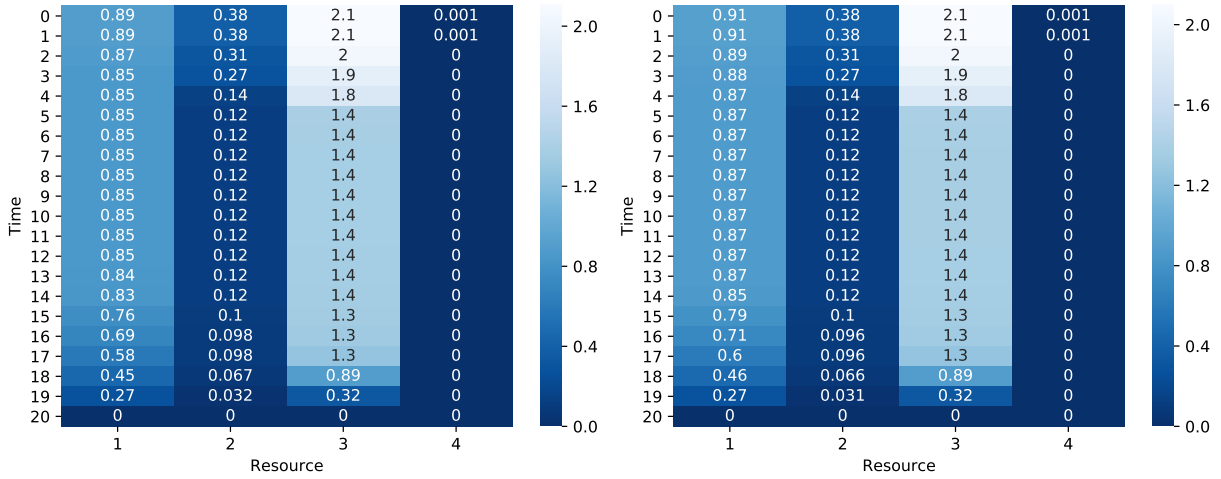
Figure 2.12: Evolution of purchased products (y-axis) via $K = 60$ iterations (x-axis).

Let us explore the resulting optimal values and offer sets in a bit more detail using two more graphs. Figure 2.13 presents the optimal values of π_{th} for every time period t and resource h. Note that the optimal values are barely changing from the second-last iteration (Figure 2.13a) to the last iteration (Figure 2.13b). In both iterations resource 4 has basically no opportunity costs, which is in line with the low demand for this product. The values for resources 1 and 2 are changing slightly, which raises the questions whether this small change even matters. Looking at Figure 2.14, it does. This figure presents the optimal offersets to be offered at any time (x-axis) and any combination of resource capacities (y-axis) in a separate colour. By "combination of resource capacities", I refer to the following: As the vector of starting capacities for resources is given by $(8, 4, 4, 8)$ and the least amount of capacity is zero, there are a total of $9 \cdot 5 \cdot 5 \cdot 9 = 2025$ combinations possible. Note however, API with linear value function approximation abstracts from these combinations, as it is not aware of the amount of available capacity. For the resulting offersets, we see that in both iteration steps, first $\{3, 5\}$ is presented to the customer and in the last time periods, $\{1, 3, 4, 5, 6\}$ is offered. But in the in-between, at the second-last iteration, $\{1, 3, 5\}$ is offered just in period 3, while for the last iteration, the transition comes via two different sets and takes longer. This figure also refers back to the identification of product 2 of not worth offering, as explained above.

eventually.

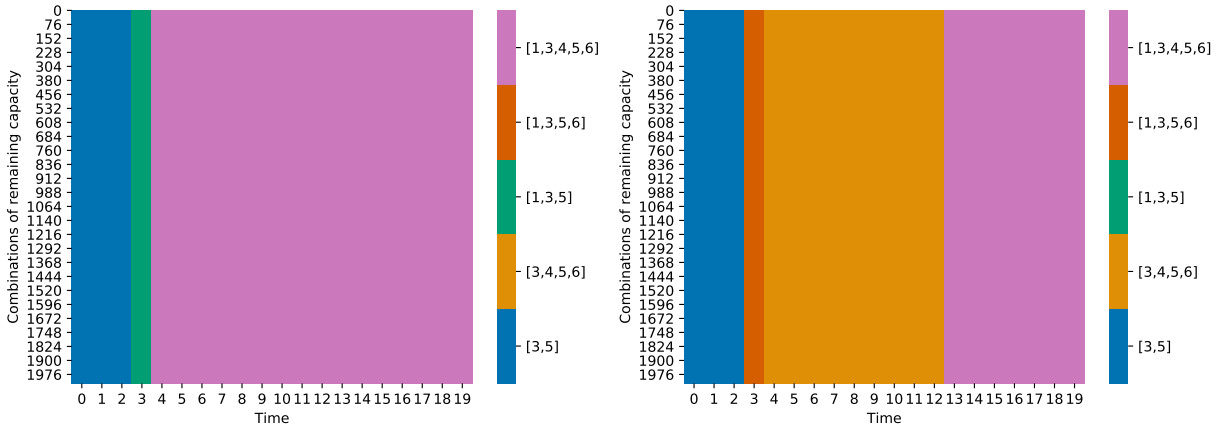
	[3,5]	[3,5,6]	[3,4,5]	[3,4,5,6]	[2]	[1,3,5]	[1,3,5,6]	[1,3,4,5,6]	[1,2,4,5]	[1,2,3,5,6]	[1,2,3,4,5,6]	[]
1	0	0	45	0	0	0	0	12929	0	20	326	2680
2	683	677	18	1987	10562	0	0	639	0	0	373	1061
3	3806	6082	0	4416	0	0	0	715	1	0	390	590
4	0	3806	0	10357	0	0	0	666	4	6	381	780
5	764	2286	0	10440	0	0	0	1343	0	8	383	776
6	0	2284	0	11134	0	0	0	1315	2	14	375	876
7	0	2284	0	11134	0	0	0	1315	2	14	375	876
8	764	1520	0	10464	0	0	0	2016	0	14	377	845
9	764	1520	0	10464	0	0	0	2016	0	14	377	845
10	0	1520	0	9756	0	764	0	2724	0	14	377	845
11	0	792	0	9389	0	796	792	3529	0	7	87	608
12	0	792	0	9406	0	1588	0	3551	0	7	87	569
13	0	0	0	8653	0	1588	792	4292	0	7	87	581
14	0	0	0	7858	0	1588	792	5070	0	7	87	598
15	0	0	0	7877	0	2380	795	4370	1	5	88	484
16	796	0	5	0	0	1584	795	10997	0	5	84	1734
17	796	795	0	8641	0	1584	0	3632	1	5	88	458
18	796	795	0	7877	0	1584	0	4394	1	5	88	460
19	796	0	0	7877	0	1584	795	4389	1	5	88	465
20	796	0	5	0	0	1584	795	10997	0	5	84	1734
21	1600	800	0	7966	0	800	0	4430	0	0	0	404
22	1600	0	0	7966	0	800	800	4430	0	0	0	404
23	800	0	0	7180	0	1600	800	5195	0	0	0	425
24	800	0	0	0	0	1600	800	11086	0	0	0	1714
25	1600	800	0	7955	0	0	800	4407	0	0	0	438
26	1600	0	0	7966	0	800	800	4430	0	0	0	404
27	1600	0	0	7966	0	800	800	4430	0	0	0	404
28	1600	0	0	4791	0	800	800	7508	0	0	0	501
29	1600	0	0	0	0	800	800	11384	0	0	0	1416
30	2400	0	0	7966	0	0	800	4430	0	0	0	404
31	1600	0	0	5591	0	800	800	6747	0	0	0	462
32	1600	0	0	0	0	800	800	11384	0	0	0	1416
33	2400	0	0	7966	0	0	800	4430	0	0	0	404
34	1600	0	0	5591	0	800	800	6747	0	0	0	462
35	1600	0	0	0	0	800	800	11384	0	0	0	1416
36	2400	0	0	7966	0	0	800	4430	0	0	0	404
37	1600	0	0	4791	0	800	800	7508	0	0	0	501
38	1600	0	0	0	0	800	800	11384	0	0	0	1416
39	2400	0	0	7966	0	0	800	4430	0	0	0	404
40	1600	0	0	6380	0	800	800	5993	0	0	0	427
41	1600	0	0	0	0	800	800	11384	0	0	0	1416
42	2400	0	0	7966	0	0	800	4430	0	0	0	404
43	1600	0	0	5591	0	800	800	6747	0	0	0	462
44	1600	0	0	0	0	800	800	11384	0	0	0	1416
45	2400	0	0	7966	0	0	800	4430	0	0	0	404
46	1600	0	0	5591	0	800	800	6747	0	0	0	462
47	1600	0	0	0	0	1600	0	11520	0	0	0	1280
48	2400	0	0	7180	0	0	800	5216	0	0	0	404
49	2400	0	0	0	0	0	800	11629	0	0	0	1171
50	2400	0	0	7966	0	0	800	4430	0	0	0	404
51	2400	0	0	4791	0	0	800	7559	0	0	0	450
52	1600	0	0	0	0	1600	0	11520	0	0	0	1280
53	2400	0	0	7180	0	0	800	5216	0	0	0	404
54	2400	0	0	0	0	800	0	11747	0	0	0	1053
55	2400	0	0	7180	0	0	800	5216	0	0	0	404
56	2400	0	0	0	0	800	0	11747	0	0	0	1053
57	2400	0	0	7180	0	0	800	5216	0	0	0	404
58	2400	0	0	0	0	800	0	11747	0	0	0	1053
59	2400	0	0	7180	0	0	800	5216	0	0	0	404
60	2400	0	0	0	0	800	0	11747	0	0	0	1053

Table 2.3: Sets offered via $K = 60$ policy iterations in the working example.



(a) $k = 59$ (b) $k = 60$

Figure 2.13: Heatmap of optimal values for π_{th} for the two final policy iterations.



(a) $k = 59$ (b) $k = 60$

Figure 2.14: Categorical map of offersets for all combinations of remaining capacity for the two final policy iterations.

```
Optimize a model with 5 rows, 64 columns and 216 nonzeros
Coefficient statistics:
  Matrix range     [6e-02, 1e+00]
  Objective range  [7e-01, 5e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [4e+00, 2e+01]
Presolve removed 0 rows and 33 columns
Presolve time: 0.00s
Presolved: 5 rows, 31 columns, 107 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    4.6288150e+02   6.226968e+01   0.000000e+00      0s
       3    9.1952381e+01   0.000000e+00   0.000000e+00      0s

Solved in 3 iterations and 0.00 seconds
Optimal objective  9.195238095e+01


------------------------------------------------------------


Optimal objective value:
 91.95238095238096

Optimal solution:
Tuple (1, 0, 0, 1, 1, 0)  is offered for a total time of  20.0


Product  1 is offered during  20.0000  time periods.
Product  2 is offered during   0.0000  time periods.
Product  3 is offered during   0.0000  time periods.
Product  4 is offered during  20.0000  time periods.
Product  5 is offered during  20.0000  time periods.
Product  6 is offered during   0.0000  time periods.
```

Figure 2.4: Optimal solution with the null set present at start.

Chapter 3

Comparison of different policies

3.1 Two sample test

Our goal is to evaluate two different policies. One policy is considered better then another policy if it generally leads to higher total revenues. To put this vague phrasing into a scientifically sound comparison, we use an approximate two sample test, which we'll describe here in more detail. We base this overview and notation mainly on Ch. 14.7 of Bamberg et al. (2011) with extensions as in Ch. 11.3 of Fahrmeir et al. (2007).

We have a total of M sample paths each consisting of T time steps, i.e. $M \times T$ random numbers determining which customer arrives and $M \times T$ random numbers leading to which product is purchased. Those two numbers are called *exogenous information*. When applying policy A, for each sample path at each time step, the respective offerset is presented to the current customer and its preferences determine which product is purchased. All purchases of one sample path are aggregated and stored, such that policy A results in the vector of values $\mathbf{v}^A \in \mathbb{R}^M$. Accordingly, $\mathbf{v}^B$ is determined. As both policies rely on the same exogenous information, the samples are dependent and a paired samples t-test has to be used.

Thus, we end up with two observations of size M

$$v_1^A, \ldots, v_M^A \text{ respectively } v_1^B, \ldots, v_M^B \ .$$

Let $\bar{\mathbf{v}^A}$ and S_A^2 resp. $\bar{\mathbf{v}^B}$ and S_B^2 be the empirical mean and empirical variance. Furthermore, μ^A and σ_A^2 denote the expected value and variance of v_i^A, respectively μ^B and σ_B^2 denote the expected value and variance of v_i^B. The hypothesis that policy A is better then policy B results in the following null and alternative hypothesis:

$$H_0 : \mu^A \leq \mu^B, H_1 : \mu^A > \mu^B$$

As $\bar{\mathbf{v}^A}$ and $\bar{\mathbf{v}^B}$ are unbiased estimators of μ^A and μ^B, the difference $D = \bar{\mathbf{v}^A} - \bar{\mathbf{v}^B}$ can be used to test the hypothesis. A large value of D represents the data to be in favour of H_1.

As we don't know the distribution of v_i^A or v_i^B and the sample size $M > 30$, we know for the test statistic

$$T = \frac{\bar{\mathbf{v}^A} - \bar{\mathbf{v}^B}}{\sqrt{\frac{S_A^2 + S_B^2}{M}}} \sim N(0; 1) \ .$$

With this knowledge, we can compute the p-value according to $p = 1 - \Phi(T)$ with $\Phi(.)$ being the standard normal distribution function. Note: The p-value can be seen as evidence against H_0. A small p-value represents strong evidence against H_0 and the null hypothesis should be rejected if the p-value is below a significance threshold α.

Chapter 4

Examples

4.1 Single-Leg Flight

For reasons of comparability, we use the same example as in Koch (2017). An airline offers four products with revenues $\boldsymbol{r} = (1000, 800, 600, 400)^T$ over $T = 400$ periods. Only one customer segment exists with arrival probability of $\lambda = 0.5$ and preference weights $\boldsymbol{u} = (0.4, 0.8, 1.2, 1.6)^T$. Different network loads can be analyzed by varying initial capacity $c^0 \in \{40, 60, \ldots, 120\}$ and varying no-purchase preference weights $u_0 \in \{1, 2, 3\}$.

4.1.1 Implementation

Here, we present the results of the exact calculation for the single leg flight example.

Storage folder:

`"C:/Users/Stefan/LRZ Sync+Share/Masterarbeit-Klein/Code/Results/singleLegFlight-Tru`

Log:

```
Time (starting): 2019-06-11 09:17:42.448064
Example :  singleLegFlight
use_var_capacities :  True


Total time needed:
 151.31108283996582 seconds =
 2.521851380666097 minutes
```

Results:

	capacity	no-purchase preference	DP-value	DP-optimal offer set
0	40	1	35952	(1, 1, 0, 0)
1	60	1	49977.1	(1, 1, 0, 0)
2	80	1	59969.8	(1, 1, 1, 0)
3	100	1	65956.4	(1, 1, 1, 0)
4	120	1	68217.2	(1, 1, 1, 1)
5	40	2	35952	(1, 1, 0, 0)
6	60	2	49977.1	(1, 1, 0, 0)
7	80	2	59969.8	(1, 1, 1, 0)
8	100	2	65956.4	(1, 1, 1, 0)
9	120	2	68217.2	(1, 1, 1, 1)
10	40	3	35952	(1, 1, 0, 0)
11	60	3	49977.1	(1, 1, 0, 0)
12	80	3	59969.8	(1, 1, 1, 0)
13	100	3	65956.4	(1, 1, 1, 0)
14	120	3	68217.2	(1, 1, 1, 1)

Furthermore, we want to visualize the value function and its approximations. We use the example of $c^0 = 60$ and $u_0 = 1$.

4.2 Multi-Leg Flight

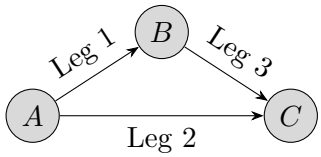
4.3 Example Bront et al. (2009)

Here, we want to present Example 0 as it was stated in Bront et al. (2009) and use this to validate our implementation (of CDLP).

Example 0 represents a small airline network as depicted by Figure 4.2a. Three cities are interconnected by three flights (legs), with capacities stated in Figure 4.2c. The booking horizon consists of $T = 30$ periods and there are five customer segments with preferences as in Figure 4.2d.

Figure 4.1: Example 0 of Bront et al. (2009) with $T = 30$ time periods.

(a) Airline network.



(b) Products.

Product	Origin-destination	Fare
1	$A \to C$	1,200
2	$A \to B \to C$	800
3	$A \to B$	500
4	$B \to C$	500
5	$A \to C$	800
6	$A \to B \to C$	500
7	$A \to B$	300
8	$B \to C$	300

(c) Resources.

Leg	Capacity
1	10
2	5
3	5

(d) Customers.

Seg.	λ_l	Consideration tuple[a]	Preference vector	No purchase preference	Description
1	0.15	$(1, 5)$	$(5, 8)$	2	Price sensitive, Nonstop (A→C)
2	0.15	$(1, 2)$	$(10, 6)$	5	Price insensitive, (A→C)
3	0.20	$(5, 6)$	$(8, 5)$	2	Price sensitive, (A→C)
4	0.25	$(3, 7)$	$(4, 8)$	2	Price sensitive, (A→B)
5	0.25	$(4, 8)$	$(6, 8)$	2	Price sensitive, (B→C)

[a]Note that in contrast to Bront et al. (2009), we use the mathematically correct terminology as *tuple* does have an inherent order, while *set* does not (and thus makes no mathematical sense to be combined with a preference vector referring to the order of products in the set).

4.3.1 CDLP

Having eight products, there is a total of $2^8 = 256$ possible offer sets[1]. Four constraints in the linear program Equation (2.13), at most four distinct sets are offered over the time horizon.

In order to reproduce the solution as stated in Bront et al. (2009) we run two very similar versions of the code, just differing in the offersets determining the variables. The optimal result given all possible offersets (i.e. also the empty set) is stated in Figure 4.3 and excluding the empty set directly from the start leads to the results presented in Figure 4.4. Note that both optimization problems terminate at the same optimal objective value, which should be the case as offering the empty set leads to 0 revenue and shouldn't be optimal.

[1]Note that in contrast to Bront et al. (2009), we include the empty set as a valid offerset because the company shall offer nothing if there is not enough capacity left to offer a product (otherwise, it is forced to deny any purchase request of a customer, leading to dissatisfaction.)

Also Gurobi becomes aware of this specific feature and presolves this column, i.e. the presolved problem consists purely of variables that are considerable. Astonishingly, even though both problems start with the same specification now (after presolving the empty set), they end up at distinct tuples that should be offered. This leads us to the insight, that Presolving in Gurobi is something different than just erasing the useless variables.

Figure 4.3: Optimal solution with the null set present at start.

```
Optimize a model with 4 rows, 256 columns and 928 nonzeros
Coefficient statistics:
  Matrix range     [4e-02, 1e+00]
  Objective range  [6e+01, 5e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [5e+00, 3e+01]
Presolve removed 0 rows and 1 columns
Presolve time: 0.00s
Presolved: 4 rows, 255 columns, 927 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    1.8638899e+05   5.936988e+02   0.000000e+00      0s
      10    1.1409091e+04   0.000000e+00   0.000000e+00      0s

Solved in 10 iterations and 0.00 seconds
Optimal objective  1.140909091e+04


------------------------------------------------------------


Optimal objective value:
 11409.090909090908

Optimal solution:
Tuple (0, 1, 1, 0, 0, 0, 0, 0)  is offered for a total time of  2.0
Tuple (1, 1, 1, 0, 0, 0, 0, 0)  is offered for a total time of  2.5455
Tuple (1, 1, 1, 0, 0, 1, 0, 0)  is offered for a total time of  25.4545


Product  1 is offered during  28.0000  time periods.
Product  2 is offered during  30.0000  time periods.
Product  3 is offered during  30.0000  time periods.
Product  4 is offered during   0.0000  time periods.
Product  5 is offered during   0.0000  time periods.
Product  6 is offered during  25.4545  time periods.
Product  7 is offered during   0.0000  time periods.
Product  8 is offered during   0.0000  time periods.
```

Figure 4.4: Optimal solution with excluding the null set before start.

```
Optimize a model with 4 rows, 255 columns and 927 nonzeros
Coefficient statistics:
  Matrix range     [4e-02, 1e+00]
  Objective range  [6e+01, 5e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [5e+00, 3e+01]
Presolve time: 0.00s
Presolved: 4 rows, 255 columns, 927 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    9.1657662e+34   5.943074e+32   9.165766e+04      0s
      10    1.1409091e+04   0.000000e+00   0.000000e+00      0s

Solved in 10 iterations and 0.00 seconds
Optimal objective  1.140909091e+04


------------------------------------------------------------


Optimal objective value:
 11409.090909090908

Optimal solution:
Tuple (0, 1, 1, 0, 0, 0, 0, 0)  is offered for a total time of  2.0
Tuple (1, 1, 1, 0, 0, 0, 0, 0)  is offered for a total time of  16.9926
Tuple (1, 1, 1, 1, 0, 1, 0, 0)  is offered for a total time of  11.0074


Product  1 is offered during  28.0000  time periods.
Product  2 is offered during  30.0000  time periods.
Product  3 is offered during  30.0000  time periods.
Product  4 is offered during  11.0074  time periods.
Product  5 is offered during   0.0000  time periods.
Product  6 is offered during  11.0074  time periods.
Product  7 is offered during   0.0000  time periods.
Product  8 is offered during   0.0000  time periods.
```

Chapter 5

Conclusion and Outlook

Chapter 6

Interesting findings

6.1 DP offer nothing at start

In the single leg flight scenario with no purchase preference of 1, we found that it makes no difference (due to rounding error), whether just the most expensive product is offered or no product at all. Thus, we want to quickly note this result down here.

We want to estimate the actions in time $t = 1$. After having the final results computed for all capacities in $t = 2$, we arrive at the following values:

Furthermore, we have the following offersets:

resulting in the purchase probabilities:

This results in the following expected values,

[**1000.**815.38461538890.90909091810.52631579955.55555556835.29411765893.33333333826.0869565**100**

i.e. offering nothing and offering just the most expensive product both result in an expected value of 1000.

As it almost never makes sense to offer no product, we switched the offer set with no products to offer at the very end. The python function `numpy.amax` returns the index of the first maximum (if there are more than one).

	value	offer_set_optimal	num_offer_set_optimal
0	0.0	0	0
1	1000.0	0	2
2	2000.0	0	2
3	3000.0	0	2
4	4000.0	0	2
5	5000.0	0	2
6	6000.0	0	2
7	7000.0	0	2
8	8000.0	0	2
9	9000.0	0	2
10	10000.0	0	2
11	11000.0	0	2
12	12000.0	0	2
13	13000.0	8	1
14	14000.0	8	1
15	15000.0	8	1
16	16000.0	8	1
17	17000.0	8	1
18	18000.0	8	1
19	19000.0	8	1
20	20000.0	8	1

	0	1	2	3
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

	0	1	2	3	4
0	0.000000	0.000000	0.000000	0.000000	1.000000
1	0.000000	0.000000	0.000000	0.307692	0.692308
2	0.000000	0.000000	0.272727	0.000000	0.727273
3	0.000000	0.000000	0.157895	0.210526	0.631579
4	0.000000	0.222222	0.000000	0.000000	0.777778
5	0.000000	0.117647	0.000000	0.235294	0.647059
6	0.000000	0.133333	0.200000	0.000000	0.666667
7	0.000000	0.086957	0.130435	0.173913	0.608696
8	0.142857	0.000000	0.000000	0.000000	0.857143
9	0.066667	0.000000	0.000000	0.266667	0.666667
10	0.076923	0.000000	0.230769	0.000000	0.692308
11	0.047619	0.000000	0.142857	0.190476	0.619048
12	0.090909	0.181818	0.000000	0.000000	0.727273
13	0.052632	0.105263	0.000000	0.210526	0.631579
14	0.058824	0.117647	0.176471	0.000000	0.647059
15	0.040000	0.080000	0.120000	0.160000	0.600000

Bibliography

Bamberg, G., Baur, F., and Krapp, M. (2011). *Statistik*. Lehr- und Handbücher der Wirtschafts- und Sozialwissenschaften. Oldenbourg, München, 16., überarb. aufl. edition.

Bertsekas, D. P. (2005). *Dynamic programming and optimal control*, volume 3 of *Athena scientific optimization and computation series*. Athena Scientific, Belmont, Mass., 3. ed. edition.

Bitran, G., Caldentey, R., and Mondschein, S. (1998). Coordinating clearance markdown sales of seasonal products in retail chains. *Operations Research*, 46(5):609–624.

Borrero, J. S., Gillen, C., and Prokopyev, O. A. (2017). Fractional 0–1 programming: applications and algorithms. *Journal of Global Optimization*, 69(1):255–282.

Bront, J. J. M., Méndez-Díaz, I., and Vulcano, G. (2009). A column generation algorithm for choice-based network revenue management. *Operations Research*, 57(3):769–784.

Domschke, W., Drexl, A., Klein, R., and Scholl, A. (2015). *Einführung in Operations Research*. Springer Gabler, Berlin and Heidelberg, 9., überarbeitete und verbesserte auflage 2015 edition.

Fahrmeir, L., Künstler, R., Pigeot, I., and Tutz, G. (2007). *Statistik: Der Weg zur Datenanalyse*. Springer-Lehrbuch. Springer, Berlin, 6., überarb. aufl. edition.

Feng, Y. and Gallego, G. (2000). Perishable asset revenue management with markovian time dependent demand intensities. *Management Science*, 46(7):941–956.

Gallego, G., Iyengar, G., Phillips, R., and A Dubey (2004). Managing flexible products on a network.

Gallego, G. and van Ryzin, G. (1997). A multiproduct dynamic pricing problem and its applications to network yield management. *Operations Research*, 45(1):24–41.

Garey, M. R. and Johnson, D. S. (ca. 2009). *Computers and intractability: A guide to the theory of NP-completeness*. A series of books in the mathematical sciences. Freeman, New York u.a, 27. print edition.

Gritzmann, P. (2013). *Grundlagen der Mathematischen Optimierung: Diskrete Strukturen, Komplexitätstheorie, Konvexitätstheorie, Lineare Optimierung, Simplex-Algorithmus, Dualität*. Aufbaukurs Mathematik. Springer, Wiesbaden.

Hansen, P., Poggi de Aragão, M. V., and Ribeiro, C. C. (1991). Hyperbolic 0–1 programming and query optimization in information retrieval. *Mathematical Programming*, 52(1-3):255–263.

Karp, R. M. (2010). Reducibility among combinatorial problems. In Jünger, M., Liebling, T. M., Naddef, D., Nemhauser, G. L., Pulleyblank, W. R., Reinelt, G., Rinaldi, G., and Wolsey, L. A., editors, *50 Years of Integer Programming 1958-2008*, pages 219–241. Springer Berlin Heidelberg, Berlin, Heidelberg.

Klein, R. (WS 2017/18). Lecture decision optmization - modellgestützte planung.

Klein, R. and Steinhardt, C. (2008). *Revenue Management: Grundlagen und mathematische Methoden*. Springer-Lehrbuch. Springer, Berlin and Heidelberg.

Koch, S. (2017). Least squares approximate policy iteration for learning bid prices in choice-based revenue management. *Computers & Operations Research*, 77:240–253.

Liu, Q. and van Ryzin, G. (2008). On the choice-based linear programming model for network revenue management. *Manufacturing & Service Operations Management*, 10(2):288–310.

Phillips, R. L. (2011). *Pricing and revenue optimization*. Stanford Business Books, Stanford, Calif., reprinted with corr edition.

Powell, W. B. (2011). *Approximate dynamic programming: Solving the curses of dimensionality*. Wiley series in probability and statistics. J. Wiley & Sons, Hoboken, N.J, 2nd ed. edition.

Prokopyev, O., Meneses, C., Oliveira, C., and Pardalos, P. (2005a). On multiple-ratio hyperbolic 0-1 programming problems. *Pacific Journal of Optimization*, 1.

Prokopyev, O. A., Huang, H.-X., and Pardalos, P. M. (2005b). On complexity of unconstrained hyperbolic 0–1 programming problems. *Operations Research Letters*, 33(3):312–318.

Strauss, A. K., Klein, R., and Steinhardt, C. (2018). A review of choice-based revenue management: Theory and methods. *European Journal of Operational Research*, 271(2):375–387.

Sutton, R. S. and Barto, A. (2018). *Reinforcement learning: An introduction*. Adaptive computation and machine learning. The MIT Press, Cambridge, MA and London, second edition edition.

Talluri, K. and van Ryzin, G. (2004). Revenue management under a general discrete choice model of consumer behavior. *Management Science*, 50(1):15–33.

Talluri, K. T. and Ryzin, G. J. (2005). *The Theory and Practice of Revenue Management*, volume 68 of *International Series in Operations Research & Management Science*. Springer Science + Business Media Inc, Boston, MA.

Train, K. (2009). *Discrete choice methods with simulation*. Cambridge University Press, Cambridge, second edition edition.

Appendix A

Notes on Code

This chapter comprises of notes on the code.

A.1 Structure of Code

1. Data Handling
    a) *A_data_entry.py*: enter data; different dictionaries for different example settings, dictionary for different variables, numpy arrays for parameter values; check dimensions; pickle data and save it on file system
    b) *A_data_read.py*: methods for reading in data of a given example name
    c) *0_settings.csv*: additional parameters that might change more often
2. Helper
    a) *B_helper.py*: methods needed by all methods
3. Exact Solution
    a) *text*

presentation_of_code

July 16, 2019

```
In [1]: from B_helper import *
        import inspect
```

1 get_offer_sets_all

1.1 Documentation

```
In [7]: print(inspect.getdoc(get_offer_sets_all))
```

```
Generates all possible offer sets, starting with offering nothing.

:param products: array of all products that can be offered.
:return: two dimensional array with rows containing all possible offer sets, starting with offering no product.
```

1.2 Source Code

```
In [8]: print(inspect.getsource(get_offer_sets_all))
```

```
def get_offer_sets_all(products):
    """
    Generates all possible offer sets, starting with offering nothing.

    :param products: array of all products that can be offered.
    :return: two dimensional array with rows containing all possible offer sets, starting with offering no product.
    """
    n = len(products)
    offer_sets_all = np.array(list(map(list, itertools.product([0, 1], repeat=n))))
    offer_sets_all = np.vstack((offer_sets_all, offer_sets_all[0]))[1:]  # move empty offer set to the end
    return offer_sets_all
```

1.3 Example

```
In [30]: products = np.arange(3)
         print("Products: \t", products)
         print("\nResult of get_offer_sets_all: \n", get_offer_sets_all(products))
```

```
Products:           [0 1 2]

Result of get_offer_sets_all:
 [[0 0 1]
 [0 1 0]
 [0 1 1]
 [1 0 0]
 [1 0 1]
```

```
[1 1 0]
[1 1 1]
[0 0 0]]
```

2 customer_choice_individual

2.1 Documentation

```
In [ ]: print(inspect.getdoc(customer_choice_individual))
```

2.2 Source Code

```
In [ ]: print(inspect.getsource(customer_choice_individual))
```

2.3 Example

```
In [ ]: offer_set_tuple = tuple(np.ones(3, dtype=np.int))
        preference_weights = np.array([1.0, 1.0, 2.0])
        preference_no_purchase = np.array(1.0)
        print("Offer set tuple: \t\t", offer_set_tuple)
        print("Preference Weights: \t\t", preference_weights)
        print("Preference no purchase: \t", preference_no_purchase)
        print("\nResult of customer_choice_individual: \n", customer_choice_individual(offer_set_tuple, preference_
```

3 customer_choice_vector

3.1 Documentation

```
In [52]: print(inspect.getdoc(customer_choice_vector))
```

```
From perspective of retailer: With which probability can he expect to sell each product (respectively non-purchase)

:param offer_set_tuple: tuple with offered products indicated by 1=product offered
:param preference_weights: preference weights of all customers
:param preference_no_purchase: preference for no purchase for all customers
:param arrival_probabilities: vector with arrival probabilities of all customer segments
:return: array with probabilities of purchase ending with no purchase

NOTE: probabilities don't have to sum up to one? BEACHTE: Unterschied zu (1) in Bront et all
```

3.2 Source Code

```
In [36]: print(inspect.getsource(customer_choice_vector))
```

```
def customer_choice_vector(offer_set_tuple, preference_weights, preference_no_purchase, arrival_probabilities):
    """
    From perspective of retailer: With which probability can he expect to sell each product (respectively non-purch

    :param offer_set_tuple: tuple with offered products indicated by 1=product offered
    :param preference_weights: preference weights of all customers
    :param preference_no_purchase: preference for no purchase for all customers
    :param arrival_probabilities: vector with arrival probabilities of all customer segments
    :return: array with probabilities of purchase for each product ending with no purchase

    NOTE: probabilities don't have to sum up to one? BEACHTE: Unterschied zu (1) in Bront et all
    """
```

```
    if sum(arrival_probabilities) > 1:
        raise ValueError("The sum of all arrival probabilities has to be <= 1.")

    probs = np.zeros(len(offer_set_tuple) + 1)
    for l in np.arange(len(preference_weights)):
        probs += arrival_probabilities[l] * customer_choice_individual(offer_set_tuple, preference_weights[l, :],
                                                                       preference_no_purchase[l])
    return probs
```

3.3 Example

```
In [41]: offer_set_tuple = tuple(np.ones(3, dtype=np.int))
         preference_weights = np.array([[1.0, 1.0, 2.0],
                                        [1.0, 2.0, 1.0],
                                         [2.0, 1.0, 1.0]])
         preference_no_purchase = np.array([1.0, 1.0, 1.0])
         arrival_probabilities = np.ones(3)/3
         print("Offer set tuple: \t\t", offer_set_tuple)
         print("Preference Weights: \n", preference_weights)
         print("Preference no purchase: \t", preference_no_purchase)
         print("Arrival probabilities: \t\t", arrival_probabilities)
         erg = customer_choice_vector(offer_set_tuple, preference_weights, preference_no_purchase, arrival_probabil
         print("\nResult of customer_choice_vector: \n", erg)
         print("Sum: ", sum(erg))
```

```
Offer set tuple:                  (1, 1, 1)
Preference Weights:
 [[1. 1. 2.]
 [1. 2. 1.]
 [2. 1. 1.]]
Preference no purchase:          [1. 1. 1.]
Arrival probabilities:                  [0.33333333 0.33333333 0.33333333]

Result of customer_choice_vector:
 [0.26666667 0.26666667 0.26666667 0.2       ]
Sum:  1.0
```

4 simulate_sales

4.1 Documentation

```
In [2]: print(inspect.getdoc(simulate_sales))
```

```
Simulates a sales event given two random numbers (customer, sale) and a offerset. Eventually, no customer arrives.
This would be the case if the random number random_customer > sum(arrival_probabilities).

:param offer_set: Products offered
:param random_customer: Determines the arriving customer (segment).
:param random_sales: Determines the product purchased by this customer.
:param arrival_probabilities: The arrival probabilities for each customer segment
:param preference_weights: The preference weights for each customer segment (for each product)
:param preferences_no_purchase: The no purchase preferences for each customer segment.
:return: The product that has been purchased. No purchase = len(products) = n   (products indexed from 0 to n-1)
```

4.2 Source Code

In [3]:
```
print(inspect.getsource(simulate_sales))
```

```
def simulate_sales(offer_set, random_customer, random_sales, arrival_probabilities, preference_weights, preferences
    """
    Simulates a sales event given two random numbers (customer, sale) and a offerset. Eventually, no customer arriv
    This would be the case if the random number random_customer > sum(arrival_probabilities).

    :param offer_set: Products offered
    :param random_customer: Determines the arriving customer (segment).
    :param random_sales: Determines the product purchased by this customer.
    :param arrival_probabilities: The arrival probabilities for each customer segment
    :param preference_weights: The preference weights for each customer segment (for each product)
    :param preferences_no_purchase: The no purchase preferences for each customer segment.
    :return: The product that has been purchased. No purchase = len(products) = n   (products indexed from 0 to n-1
    """
    customer = random_customer <= np.array([*np.cumsum(arrival_probabilities), 1.0])
    customer = min(np.array(range(0, len(customer)))[customer])

    if customer == len(arrival_probabilities):
        return len(preference_weights[0])  # no customer arrives => no product sold (product out of range)
    else:
        product = random_sales <= np.cumsum(customer_choice_individual(offer_set,
                                                                       preference_weights[customer],
                                                                       preferences_no_purchase[customer]))
        product = min(np.arange(len(preference_weights[0]) + 1)[product])
        return product
```

4.3 Example

In [26]:
```
offer_set_tuple = tuple(np.ones(3, dtype=np.int))
preference_weights = np.array([[1.0, 1.0, 2.0],
                               [1.0, 2.0, 1.0],
                                [2.0, 1.0, 1.0]])
preference_no_purchase = np.array([1.0, 1.0, 1.0])
arrival_probabilities = np.ones(3)/4

print("Offer set tuple: \t\t", offer_set_tuple)
print("Preference Weights: \n", preference_weights)
print("Preference no purchase: \t", preference_no_purchase)
print("Arrival probabilities: \t\t", arrival_probabilities)

print("\n   Customer preferences of customer 0: \t", customer_choice_individual(offer_set_tuple, preferenc
print("   Customer preferences of customer 1: \t", customer_choice_individual(offer_set_tuple, preference_

random_customer = 0.1
random_sales = 0.1
print("\n\n##########\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.1
random_sales = 0.3
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
```

```
print("Result of simulate_sales: \n", erg)

random_customer = 0.1
random_sales = 0.5
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.1
random_sales = 0.9
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.4
random_sales = 0.1
print("\n\n##########\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.4
random_sales = 0.3
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.4
random_sales = 0.5
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.4
random_sales = 0.9
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.8
random_sales = 0.1
print("\n\n##########\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.8
random_sales = 0.3
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.8
```

```
        random_sales = 0.5
        print("\n\nRandom customer: \t", random_customer)
        print("Random sales: \t", random_sales)
        erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
        print("Result of simulate_sales: \n", erg)

        random_customer = 0.8
        random_sales = 0.9
        print("\n\nRandom customer: \t", random_customer)
        print("Random sales: \t", random_sales)
        erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
        print("Result of simulate_sales: \n", erg)
```

```
Offer set tuple:                 (1, 1, 1)
Preference Weights:
 [[1. 1. 2.]
 [1. 2. 1.]
 [2. 1. 1.]]
Preference no purchase:          [1. 1. 1.]
Arrival probabilities:                  [0.25 0.25 0.25]

   Customer preferences of customer 0:          [0.2 0.2 0.4 0.2]
   Customer preferences of customer 1:          [0.2 0.4 0.2 0.2]


#########
Random customer:          0.1
Random sales:          0.1
Result of simulate_sales:
 0


Random customer:          0.1
Random sales:          0.3
Result of simulate_sales:
 1


Random customer:          0.1
Random sales:          0.5
Result of simulate_sales:
 2


Random customer:          0.1
Random sales:          0.9
Result of simulate_sales:
 3


#########
Random customer:          0.4
Random sales:          0.1
Result of simulate_sales:
 0


Random customer:          0.4
Random sales:          0.3
```

```
Result of simulate_sales:
 1


Random customer:          0.4
Random sales:          0.5
Result of simulate_sales:
 1


Random customer:          0.4
Random sales:          0.9
Result of simulate_sales:
 3


#########
Random customer:          0.8
Random sales:          0.1
Result of simulate_sales:
 3


Random customer:          0.8
Random sales:          0.3
Result of simulate_sales:
 3


Random customer:          0.8
Random sales:          0.5
Result of simulate_sales:
 3


Random customer:          0.8
Random sales:          0.9
Result of simulate_sales:
 3
```

5 calc_value_marginal

5.1 Documentation

```
In [2]: print(inspect.getdoc(calc_value_marginal))
```

```
Calculates the marginal value as indicated at Bront et al, 4.2.2 Greedy Heuristic -> step 4a

:param indices_inner_sum: C_l intersected with (S union with {j})
:param pi: vector of dual prices for each resource (np.inf := no capacity)
:param revenues: vector of revenue for each product
:param A: matrix with resource consumption of each product (one row = one resource)
:param arrival_probabilities: The arrival probabilities for each customer segment
:param preference_weights: The preference weights for each customer segment (for each product)
:param preferences_no_purchase: The no purchase preferences for each customer segment.
:return: The value inside the argmax (expected marginal value given one set of products to offer)
```

5.2 Source Code

In [2]:
```
print(inspect.getsource(calc_value_marginal))
```

```
def calc_value_marginal(indices_inner_sum, pi, revenues, A, arrival_probabilities, preference_weights, preferences_
    """
    Calculates the marginal value as indicated at Bront et al, 4.2.2 Greedy Heuristic -> step 4a

    :param indices_inner_sum: C_l intersected with (S union with {j})
    :param pi: vector of dual prices for each resource (np.inf := no capacity)
    :param revenues: vector of revenue for each product
    :param A: matrix with resource consumption of each product (one row = one resource)
    :param arrival_probabilities: The arrival probabilities for each customer segment
    :param preference_weights: The preference weights for each customer segment (for each product)
    :param preferences_no_purchase: The no purchase preferences for each customer segment.
    :return: The value inside the argmax (expected marginal value given one set of products to offer)
    """
    v_temp = 0
    for l in np.arange(len(preference_weights)):  # sum over all customer segments
        v_temp += arrival_probabilities[l] * \
                  sum(indices_inner_sum * (revenues - np.apply_along_axis(sum, 1, A.T * pi)) *
                      preference_weights[l, :]) / \
                  (sum(indices_inner_sum * preference_weights[l, :]) + preferences_no_purchase[l])
    return v_temp
```

5.3 Example

In [16]:
```
print("Example analogously to Bront et al\n\n")
revenues = np.array([100, 19, 19])
pi = np.array([0])
A = np.array([[1, 1, 1]])
arrival_probabilities = np.array([1, 1, 1])
preference_weights = np.array([[1,1,1],
                               [0,1,0],
                               [0,0,1]])
preferences_no_purchase = np.array([1,1,1])

print("Revenues: \t\t\t", revenues)
print("Pi: \t\t\t\t", pi)
print("A:\t\t\t\t", A)
print("Preference Weights: \n", preference_weights)
print("Preference no purchase: \t", preferences_no_purchase)
print("Arrival probabilities: \t\t", arrival_probabilities)

indices_inner_sum = np.array([1, 0, 0])
print("\n#####\nIndices inner sum: \t", indices_inner_sum)
erg = calc_value_marginal(indices_inner_sum, pi, revenues, A, arrival_probabilities, preference_weights, p
print("Result of simulate_sales: \n", erg)

indices_inner_sum = np.array([1, 1, 1])
print("\n#####\nIndices inner sum: \t", indices_inner_sum)
erg = calc_value_marginal(indices_inner_sum, pi, revenues, A, arrival_probabilities, preference_weights, p
print("Result of simulate_sales: \n", erg)
```

```
Example analogously to Bront et al


Revenues:                       [100  19  19]
```

```
Pi:                                  [0]
A:                                [[1 1 1]]
Preference Weights:
 [[1 1 1]
 [0 1 0]
 [0 0 1]]
Preference no purchase:          [1 1 1]
Arrival probabilities:                 [1 1 1]

#####
Indices inner sum:          [1 0 0]
Result of simulate_sales:
 50.0

#####
Indices inner sum:          [1 1 1]
Result of simulate_sales:
 53.5
```

6 determine_offer_tuple

6.1 Documentation

```
In [17]: print(inspect.getdoc(determine_offer_tuple))
```

```
Determines the offerset given the bid prices for each resource.

Implement the Greedy Heuristic from Bront et al: A Column Generation Algorithm ... 4.2.2
and extend it for the epsilon greedy strategy
:param pi: vector of dual prices for each resource (np.inf := no capacity)
:param eps: epsilon value for epsilon greedy strategy (eps = 0 := no greedy strategy to apply)
:param revenues: vector of revenue for each product
:param A: matrix with resource consumption of each product (one row = one resource)
:param arrival_probabilities: The arrival probabilities for each customer segment
:param preference_weights: The preference weights for each customer segment (for each product)
:param preferences_no_purchase: The no purchase preferences for each customer segment.
:return: the offer set to be offered
```

6.2 Source Code

```
In [24]: print(inspect.getsource(determine_offer_tuple))
```

```
def determine_offer_tuple(pi, eps, revenues, A, arrival_probabilities, preference_weights, preferences_no_purchase)
    """
    Determines the offerset given the bid prices for each resource.

    Implement the Greedy Heuristic from Bront et al: A Column Generation Algorithm ... 4.2.2
    and extend it for the epsilon greedy strategy
    :param pi: vector of dual prices for each resource (np.inf := no capacity)
    :param eps: epsilon value for epsilon greedy strategy (eps = 0 := no greedy strategy to apply)
    :param revenues: vector of revenue for each product
    :param A: matrix with resource consumption of each product (one row = one resource)
    :param arrival_probabilities: The arrival probabilities for each customer segment
    :param preference_weights: The preference weights for each customer segment (for each product)
    :param preferences_no_purchase: The no purchase preferences for each customer segment.
    :return: the offer set to be offered
    """
```

```
# no resources left => nothing to be sold
if all(pi == np.inf):
    return tuple(np.zeros_like(revenues))

# epsilon greedy strategy - offer no products
eps_prob = random.random()
if eps_prob < eps/2:
    return tuple(np.zeros_like(revenues))

# epsilon greedy strategy - offer all products
if eps_prob < eps:
    offer_tuple = np.ones_like(revenues)
    offer_tuple[np.sum(A[[pi == np.inf], :], axis=0) > 0] = 0  # one resource not available => don't offer prod
    return tuple(offer_tuple)

# setup
offer_tuple = np.zeros_like(revenues)

# line 1
s_prime = revenues - np.apply_along_axis(sum, 1, A.T * pi) > 0
if all(np.invert(s_prime)):
    return tuple(offer_tuple)

# line 2-3
# offer_sets_to_test has in each row an offer set, we want to test
offer_sets_to_test = np.zeros((sum(s_prime), len(revenues)))
offer_sets_to_test[np.arange(sum(s_prime)), np.where(s_prime)] = 1
offer_sets_to_test += offer_tuple
offer_sets_to_test = (offer_sets_to_test > 0)

value_marginal = np.apply_along_axis(calc_value_marginal, 1, offer_sets_to_test, pi, revenues,
                                     preference_weights, arrival_probabilities, A, preferences_no_purchase)

offer_tuple[np.argmax(value_marginal)] = 1
s_prime = s_prime & offer_tuple == 0
v_s = np.amax(value_marginal)

# line 4
while True:
    # 4a
    # offer_sets_to_test has in each row an offer set, we want to test
    offer_sets_to_test = np.zeros((sum(s_prime), len(revenues)))
    offer_sets_to_test[np.arange(sum(s_prime)), np.where(s_prime)] = 1
    offer_sets_to_test += offer_tuple
    offer_sets_to_test = (offer_sets_to_test > 0)

    # 4b
    value_marginal = np.apply_along_axis(calc_value_marginal, 1, offer_sets_to_test, pi, revenues,
                                         preference_weights, arrival_probabilities, A, preferences_no_purchase)

    if np.amax(value_marginal) > v_s:
        v_s = np.amax(value_marginal)
        offer_tuple[np.argmax(value_marginal)] = 1
        s_prime = s_prime & offer_tuple == 0
        if all(offer_tuple == 1):
            break
    else:
        break
```

```
    return tuple(offer_tuple)
```

6.3 Example

```
In [23]: print("Example analogously to Bront et al\n\n")
         revenues = np.array([100, 19, 19])
         pi = np.array([0])
         A = np.array([[1, 1, 1]])
         arrival_probabilities = np.array([1, 1, 1])
         preference_weights = np.array([[1,1,1],
                                        [0,1,0],
                                        [0,0,1]])
         preferences_no_purchase = np.array([1,1,1])
         eps = 0

         print("Revenues: \t\t\t", revenues)
         print("Pi: \t\t\t\t", pi)
         print("A:\t\t\t\t", A)
         print("Preference Weights: \n", preference_weights)
         print("Preference no purchase: \t", preferences_no_purchase)
         print("Arrival probabilities: \t\t", arrival_probabilities)
         print("Epsilon: \t\t\t", eps)

         erg = determine_offer_tuple(pi, eps, revenues, A, arrival_probabilities, preference_weights, preferences_n
         print("\nResult of determine_offer_tuple: \n", erg)
```

```
Example analogously to Bront et al


Revenues:                           [100  19  19]
Pi:                                   [0]
A:                                  [[1 1 1]]
Preference Weights:
 [[1 1 1]
 [0 1 0]
 [0 0 1]]
Preference no purchase:           [1 1 1]
Arrival probabilities:                [1 1 1]
Epsilon:                           0

Result of determine_offer_tuple:
 (1, 0, 0)
```

Appendix C

Mathematical Theory and Proofs

Lemma 1. If the optimal solution of the policy iteration optimization model as presented in Equations (2.51) to (2.56) is unique for each policy iteration k, then exponential smoothing results in the same return variable as using the optimal updates in each iteration and returning the overall average in the end. Exponential smoothing was presented in Equations (2.59) and (2.60) and is shown again without the t index for convenience.

$$\theta^{k+1} = \left(1 - \frac{1}{k}\right)\theta^k + \frac{1}{k}\theta^{update,k} \tag{C.1}$$

$$\boldsymbol{\pi}^{k+1} = \left(1 - \frac{1}{k}\right)\boldsymbol{\pi}^k + \frac{1}{k}\boldsymbol{\pi}^{update,k} \tag{C.2}$$

The other procedure of using the updates directly was presented in Equations (2.57) to (2.58) and is also shown again.

$$\theta^{k+1} = \theta^{update,k} \tag{C.3}$$

$$\boldsymbol{\pi}^{k+1} = \boldsymbol{\pi}^{update,k} \tag{C.4}$$

Proof. Without loss of generality (w. l. o. g.), it suffices to show the statement just for θ. We have to show that the iterative exponential smoothing results in the same value as taking the average in the very end. We proof this by induction over the total number of policy iterations K. The statement trivially holds for $K = 1$. So let the statement hold for $K = n$, i.e.

$$\theta^{n+1} = \left(1 - \frac{1}{n}\right)\theta^n + \frac{1}{n}\theta^{update,n} = \frac{1}{n}\sum_{i=1}^{n}\theta^{update,i} \ .$$

Now, we apply one more step of exponential smoothing, use the induction hypothesis and

summation rules to derive our hypothesis and proof Lemma 1.

$$\theta^{n+2} = \left(1 - \frac{1}{n+1}\right)\theta^{n+1} + \frac{1}{n+1}\theta^{update,n+1} \tag{C.5}$$

$$= \left(1 - \frac{1}{n+1}\right)\left(\frac{1}{n}\sum_{i=1}^{n}\theta^{update,i}\right) + \frac{1}{n+1}\theta^{update,n+1} \tag{C.6}$$

$$= \frac{n}{n+1}\frac{1}{n}\sum_{i=1}^{n}\theta^{update,i} + \frac{1}{n+1}\theta^{update,n+1} \tag{C.7}$$

$$= \frac{1}{n+1}\sum_{i=1}^{n}\theta^{update,i} + \frac{1}{n+1}\theta^{update,n+1} \tag{C.8}$$

$$= \frac{1}{n+1}\sum_{i=1}^{n+1}\theta^{update,i} \tag{C.9}$$

□

Remark 2. Note that the optimization problem depends on the current values of the optimization variables as these are used in the sample run to determine the offersets and ultimately affect the sample value function $\hat{V}_t^i$. So theoretically that there is no (obvious) reason why Lemma 1 shall be applicable. But in my experiments it made no difference, whether exponential smoothing was applied directly or the average taken in the end. The results in this thesis were generated by using exponential smoothing.

Appendix D

Data Scientist

Figure D.1: Certificate "Data Scientist with Python Track" with 100 learning hours.

This career track on DataCamp helped me to improve my Python coding skills and apply analytical techniques in Python. It comprised the courses:

1. Introduction to Python
2. Intermediate Python for Data Science
3. Python Data Science Toolbox (Part 1)
4. Python Data Science Toolbox (Part 2)
5. Importing Data in Python (Part 1)
6. Importing Data in Python (Part 2)
7. Cleaning Data in Python
8. pandas Foundations
9. Manipulating DataFrames with pandas
10. Merging DataFrames with pandas
11. Analyzing Police Activity with pandas
12. Intro to SQL for Data Science
13. Introduction to Relational Databases in SQL
14. Introduction to Data Visualization with Python
15. Interactive Data Visualization with Bokeh
16. Statistical Thinking in Python (Part 1)
17. Statistical Thinking in Python (Part 2)
18. Joining Data in SQL
19. Introduction to Shell for Data Science
20. Conda Essentials
21. Supervised Learning with scikit-learn
22. Machine Learning with the Experts: School Budgets
23. Unsupervised Learning in Python
24. Machine Learning with Tree-Based Models in Python
25. Deep Learning in Python
26. Network Analysis in Python (Part 1)

Appendix E

Schriftliche Versicherung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbständig und ohne fremde Hilfe verfasst wurde. Alle Stellen, die ich wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Quellen übernommen habe, habe ich als solche gekennzeichnet. Es wurden alle Quellen, auch Internetquellen, ordnungsgemäß angegeben.

Augsburg	30.09.2019
Ort	Datum

Stefan Glogger