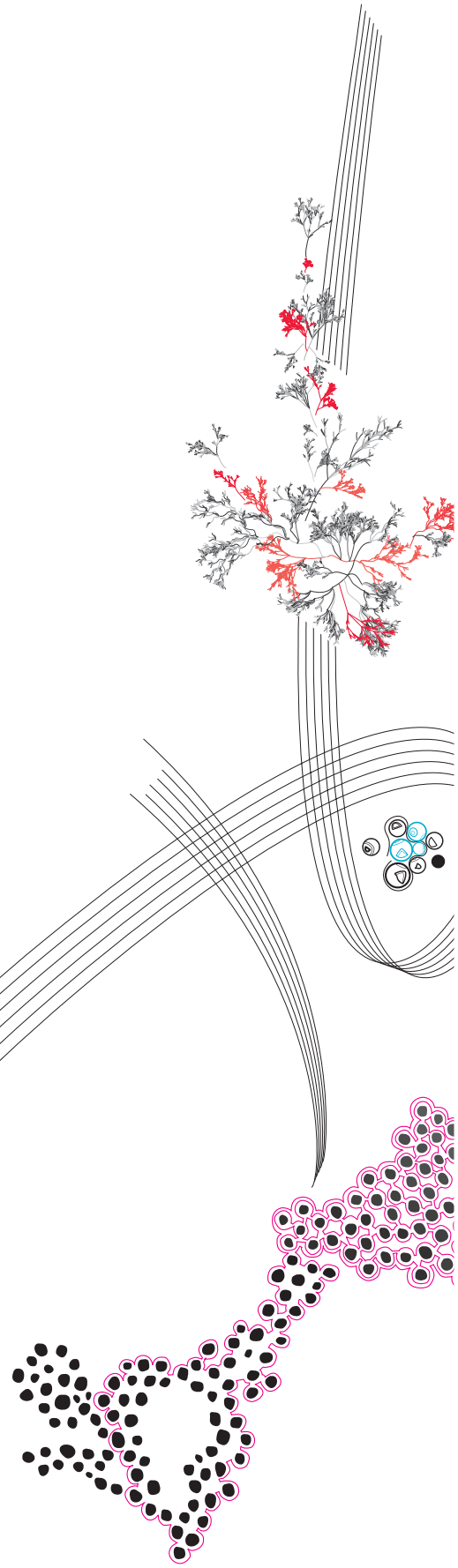

BSc Thesis Applied Mathematics

Operations Research and Airlines: Solving the Meal Provisioning Problem

Marieke Mimi Cato Romeijn

Supervisor: N. M. van Dijk

June, 2021

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

This paper was written as my Bachelor Thesis for the Bachelor Applied Mathematics at the University of Twente. I want to thank my supervisor Nico M. van Dijk for guiding me through this process and always being there when I needed help.

Operations Research and Airlines: Solving the Meal Provisioning Problem

Marieke M. C. Romeijn*

June, 2021

Abstract

This paper investigates the problem of meal provisioning on airplanes. Usually, there are several moments that an airline can order meals for a flight, and as the day of the flight comes closer, the price of the meals increase. On the other hand, as the day of the flight gets closer, the airline has more information about how many tickets will be solved. Using Operations Research methods such as Monte Carlo simulation and analytically (Newsboy problem), we will solve the simplified problem of only having one decision moment. Then we will focus on the main problem of having three decision moments. We will solve this using Stochastic Dynamic Programming (SDP). For all methods, Python is used to program and get results.

The Monte Carlo simulation and the Newsboy approach both came to the same conclusions, which is very strong. The SDP gave us different results, as this method has multiple decision moments.

Keywords: Meal provisioning, operations research, Markov decision problem, newsboy problem, Monte Carlo simulation, Markov decision tree, stochastic dynamic programming

*Email: m.m.c.romeijn@student.utwente.nl

Contents

1 Introduction **1**

1.1 Problem Description 1

1.2 Pilot data 1

1.3 Methods 2

2 Single decision moment **3**

2.1 Monte Carlo Simulation 3

2.1.1 Results 3

2.2 Newsboy Problem: Analytic Approach 4

2.2.1 Results 6

3 Multiple decision moments **8**

3.1 Markov Decision Tree 8

3.2 Stochastic Dynamic Programming 11

3.2.1 Results 13

4 Results **14**

References **15**

Appendices **16**

A Monte Carlo Simulation Code **16**

B Newsboy Code **18**

C Stochastic Dynamic Programming Code **21**

D Probability Distribution Function **27**

1 Introduction

1.1 Problem Description

Passengers on a plane expect the best service while they are in the air. They all want to have access to drinks, and on longer flights, they of course expect a meal. Catering meals for a flight is a difficult process, since the airline does not want to order too little or too many meals. When ordering too little meals, the additional meals need to be brought to the plane last minute, which is very expensive. When ordering too many meals, these remaining meals have been paid for already while there is no use for them. Furthermore, there are multiple moments in which meals can be ordered, from months prior to the flight up to a day prior to the flight. The meals get more expensive as the day of the flight gets closer, so we want to find a way to order meals in order to minimise the expected costs while satisfying demands. Since there is no data available, we will study the problem with fictional pilot data.

1.2 Pilot data

Suppose that we have a flight with 200 seats. The costs of a meal are €25,- three months, €40,- two weeks and €75,- one day prior to the flight. In addition, when there are meals short one hour before the flight itself, it not only costs €100,- per meal, but also €200,- for the delivery of the meals, independent of the number of the meals.

We used theoretical data to estimate the optimal order at any moment that seems reasonable. At two weeks before the flight, the following distribution will be used (Table 1).

TABLE 1: Probabilities two weeks before the flight.

# tickets sold	0	40	80	120	160	200
Probability	.05	.20	.25	.30	.15	.05

At one day before the flight, an extra 10, 20, or 30 tickets can be sold with the probabilities in Table 2.

TABLE 2: Probabilities one day before the flight.

# tickets sold	0	10	20	20
Probability	.10	.40	.30	.20

Finally, some additional tickets can be sold the day of the flight. Those tickets are sold according to the probabilities presented in Table 3.

TABLE 3: Probabilities one hour prior to the flight.

# tickets sold	0	1	2	3	4	5	6	7	8	9
Probability	.05	.10	.15	.15	.15	.10	.10	.10	.05	.05

This can be visualised in the simplified diagram shown below (Figure 1).

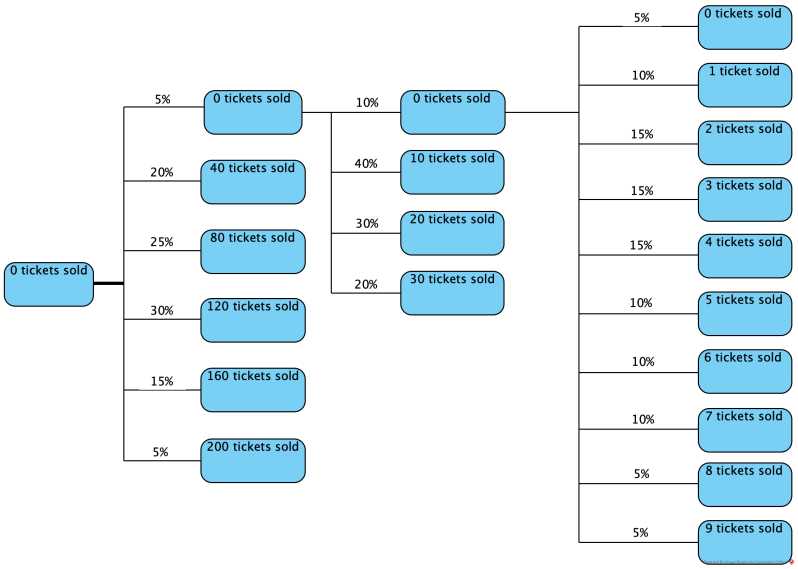

FIGURE 1: Visualised data

1.3 Methods

In an other paper, a similar problem has been studied ([2]). In this article, this problem is solved using Stochastic Dynamic Programming. In this paper we will simplify the problem by first only looking at one decision moment, and building it up to the complete problem. To do this, we will incorporate multiple methods that are used in Operations Research. We will work with Monte Carlo simulation, the Newsboy Problem and finally Stochastic Dynamic Programming.

We begin with simplifying the problem so that we only have one decision moment three months prior to the flight. This problem will be solved in two ways, namely Monte Carlo simulation and by using the Newsboy Problem. First, we will focus on Monte Carlo simulation (see Section 2.1). Next, we will move on to the Newsboy Problem approach, in which we will solve the problem analytically (see Section 2.2). Then we will add the remaining decision moments two weeks and one day ahead of the flight to solve the given problem using Stochastic Dynamic Programming (SDP) (see Section 3). There, we will start by visualising the problem using a Markov Decision Tree (Section 3.1), and finally we will solve the problem by setting up the SDP (Section 3.2). For all approaches, we used Python to code. The code can be found in the Appendix (see Appendix A, B, and C).

2 Single decision moment

2.1 Monte Carlo Simulation

The first method we will use is Monte Carlo simulation. We will use Monte Carlo simulation to solve the simple problem of having only one decision moment three months prior to the flight. Monte Carlo simulation can best be described as throwing a large number of dice for any number of meals that can be ordered (0 to 200) to generate random numbers (in this case the number of tickets sold) and generating the costs that will be made. The optimum number of meals to order will be the number n^* for which the average resulted in the lowest costs. We can describe the simulation in Algorithm 1, where k is the number of generated test values (or dice thrown).

Algorithm 1: Monte Carlo Simulation

```
for i = 0 to 200 do
    tickets_sold = k random numbers according to a probability distribution;
    for j = 0 to k do
        costs[i][j] = costs of tickets_sold[j] when ordering i meals;
    end
end
mean = mean of costs over experiments per # meals ordered// array of length 201
optimum = index of the minimum over mean;
```

2.1.1 Results

To start with, we will use an example in which the demand three months before the flight can be estimated by a normal distribution with $\mu = 150$ and $\sigma = 30$. We choose $k = 1,000,000$, since this gives us consistent results. We find that $n^* = 172$, where the 95% confidence interval of average expected costs is €$(4692.388, 4692.391)$. This means that the population mean of the costs when ordering 172 meals is in this interval with probability 95%. In Figure 2, the average costs of ordering 167 to 177 are shown. We see here that the costs of ordering 171, 172, 173 and 174 meals is very close to each other.

We will now focus on the given problem. Instead of the normal distribution, we will use the distributions given in Tables 1, 2, and 3. By combining these distribution, we can make a new distribution that represents the number of tickets sold one hour before the flight, when we know nothing yet at three months prior to the flight. The values of the probability distribution function can be found in Table 6 in Appendix D.

Now, we will choose $k = 10,000,000$, as this brings us consistent results. We find that $n^* = 153$, where the 95% confidence interval of average expected costs is €$(4524.1384, 4524.1389)$. The average costs of ordering 148 to 158 meals is shown again in Figure 3, where we see that the costs of ordering 153, 154, or 155 meals are again relatively close to each other.

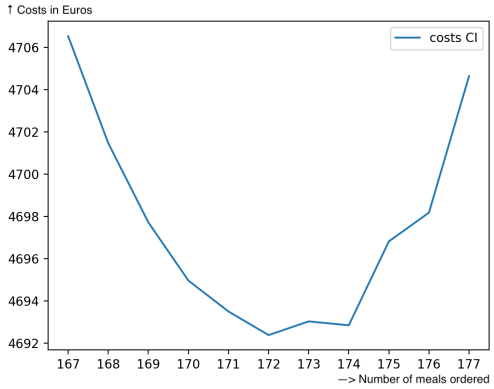

FIGURE 2: Costs when using N(150, 30) distribution.

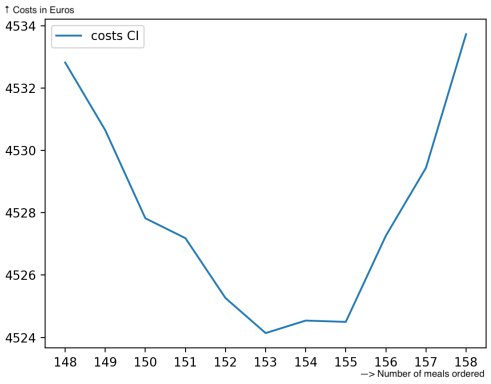

FIGURE 3: Costs when using combined distribution.

2.2 Newsboy Problem: Analytic Approach

The second method is in known in literature as the Newsboy Problem ([3]), it takes an analytic approach. We will make a function of expected costs $z(n)$ when ordering n meals, and we will find the optimum number of meals n^* so that the expected costs are minimised.

Let $z_t(n)$ be the expected costs depending on number of ordered meals n and moment t, and let c be the total costs. Furthermore, let $c_{u,n,q}$ be the costs made for ordering n meals while the demand is q and $q > n$, so there are too few meals ordered (undercosts). Let $c_{o,n,q}$ be the costs made for ordering n meals while the demand is q and $q \leq n$, so there are too many meals ordered (overcosts). Let c_t be the costs for a meal at moment t, $t = 1, 2, 3$ ($t = 1$ three months, $t = 2$ two weeks, and $t = 3$ one day prior to the flight). So $c_1 = 25, c_2 = 40, c_3 = 75$. Finally, let Q be the random variable representing the demand of meals with probability density function $f(q)$ and cumulative distribution function $F(q)$. It follows that $\lim_{q\to\infty} F(q) = 1$. Then

$$
\begin{aligned}
c &= \begin{cases} c_{u,n,q} & \text{if } q > n,\ q \leq 200, \\ c_{o,n,q} & \text{if } q \leq n,\ q \leq 200, \\ c_{e,n,q} & \text{if } q > 200, \end{cases} \\
&= \begin{cases} c_t \cdot n + 200 + 100(q-n) & \text{if } q > n,\ q \leq 200, \\ c_t \cdot n & \text{if } q \leq n,\ q \leq 200 \\ c_t \cdot n + 200 + 100(200-n) & \text{if } q > 200. \end{cases}
\end{aligned}
$$

Next, we can calculate the expected costs $z_t(n)$.

$$
\begin{aligned}
z_t(n) &= E[c] \\
&= \int_0^{\infty} cf(q)dq \\
&= \int_0^{200} cf(q)dq + \int_{200}^{\infty} (c_i \cdot n + 200 + 100(200-n))f(q)dq \\
&= \int_0^{n} c_{o,n,q} f(q)dq + \int_n^{200} c_{u,n,q} f(q)dq + c_t n \int_{200}^{\infty} f(q)dq \\
&\qquad +(200 + 100(200-n)) \int_{200}^{\infty} f(q)dq \\
&= \int_0^{n} c_t n f(q)dq + \int_n^{200} (c_t n + 200 + 100(q-n))f(q)dq + c_t n \int_{200}^{\infty} f(q)dq \\
&\qquad +(200 + 100(200-n))(\lim_{x\to\infty} F(x) - F(200)) \\
&= c_t n \int_0^{n} f(q)dq + c_t n \int_n^{200} f(q)dq + 100\big[(2-n)\int_n^{200} f(q)dq + \int_n^{200} qf(q)dq \\
&\qquad +c_t n \int_{200}^{\infty} f(q)dq + (2 + (200-n))(1 - F(200))\big] \\
&= c_t n \int_0^{\infty} f(q)dq + 100\big[(2-n)\int_n^{200} f(q)dq + \int_n^{200} qf(q)dq \\
&\qquad +(202-n)(1-F(200))\big]. \qquad (1)
\end{aligned}
$$

We need to find the optimal value of n to minimise z. We will do this by taking the derivative of Equation 1.

$$
\begin{aligned}
\frac{dz_t}{dn} =& c_t \int_0^{\infty} f(q)dq + 100\big[- \int_n^{200} f(q)dq + (2-n)\cdot -f(n) - nf(n) \\
&\qquad -(1-F(200))\big] \\
=& c_t(\lim_{x\to\infty} F(x) - F(0)) + 100\big[F(n) - 2f(n) - 1\big] \\
=& c_t(1 - F(0)) + 100\big[F(n) - 2f(n) - 1\big]
\end{aligned}
$$

Let n^* be the optimum number of meals to order such that the expected costs are minimised. Then we need to find n^* such that

$$
c_t(1 - F(0)) + 100\big[F(n^*) - 2f(n^*) - 1\big] = 0. \qquad (2)
$$

Remark: Note that Equation 2 can be used for any continuous or discrete probability distribution. This makes the Newsboy approach powerful, as it can be used for any similar problem.

2.2.1 Results

First, just as in Section 2.1, we let $Q \sim N(150, 30)$. Using Python, we get $n^* = 172$ for $t = 1$ (so $c_1 = 25$) (see the code in Appendix B). This is expected, as it was also found in Section 2.1 when using Monte Carlo simulation.

Next, we again use the distribution that combines the distributions given in Tables 1, 2, and 3. The values of the probability distribution function can be found in Table 6 in Appendix D.

Using Python (see Appendix B), we find $n^* = 153$ for $t = 1$ (so $c_1 = 25$), which is again the same as found in Section 2.1. This result is very powerful. We can calculate the same result when using simulation as when we use an analytical approach, but when using the analytical approach, we can get to this result quicker. The simulation takes longer as it needs a lot of generated values to get an accurate result.

Now that we have a result for three months ahead, we can also find the optimal order two weeks in advance. Two weeks in advance, we already know more than three months prior to the flight. At that point, we know that there can be 0, 40, ..., or 200 tickets sold, so we can work with that. We will use $c_2 = 40$, and we will now look at the case that there 0 tickets sold at this moment. Now, we can combine the distributions from 2 and 3 to create a distribution that will tell us how many tickets will be sold at the time of the flight. This probability distribution function can be found in Table 7.

When using Python (see Appendix B), we now find that it is best to order $n^* = 24$. We can do the same in the situation where there are 40, 80, ..., or 200 tickets sold at $t = 2$, or we can go one step further to $t = 3$.

One day before the flight ($t = 3$), we know that the number of tickets sold is 0, 10, ... or 200. We can use the distribution in Table 3 and we let $c_3 = 75$. We can combine all findings in the Table 4. At any t, we can look up the current demand. Below the current demand, the optimal order and the expected costs from that point on including the current order when ordering the given number of meals are shown.

TABLE 4: Optimal orders and their expected costs according to the Newsboy approach.

t = 1	Demand	0					
	Optimal order	153					
	Expected costs [€]	4522,70					
t = 2	Demand	0	40	80	120	160	200
	Optimal order	24	64	104	144	184	200
	Expected costs [€]	1257,50	2857,50	4457,50	6057,49	7657,5	8000
t = 3	Demand	0	10	20	30	40	50
	Optimal order	4	14	24	34	44	54
	Expected costs [€]	485	1235	1985	2735	3485	4235
	Demand	60	70	80	90	100	110
	Optimal order	64	74	84	94	104	114
	Expected costs [€]	4985	5735	6485	7235	7985	8735
	Demand	120	130	140	150	160	170
	Optimal order	124	134	144	154	164	174
	Expected costs [€]	9485	10235	10985	11735	12485	13235
	Demand	180	190	200			
	Optimal order	184	194	200			
	Expected costs [€]	13985	14735	15000			

3 Multiple decision moments

We will solve the problem using Stochastic Dynamic Programming ([5]). This allows us to have multiple decision moments. The SDP calculations do not take long when there are just three decision moments, but the time will grow fairly quickly when there are more moments. First, we will show a decision tree that describes the structure of the problem.

3.1 Markov Decision Tree

A Markov Decision Tree (MDT) is a tool to show the structure of the problem. The blue boxes show the decisions that can be made. Those decisions can be made at the grey triangles. Finally, the pink circles show the possible number of tickets sold with their corresponding probabilities.

Below in Figure 4, an overview of the structure of the problem is shown. A few paths are shown, to not make it too big. We see that the complexity grows very quickly, as there are multiple decisions at any decision moment, and there are multiple possibilities of sold tickets at any moment.

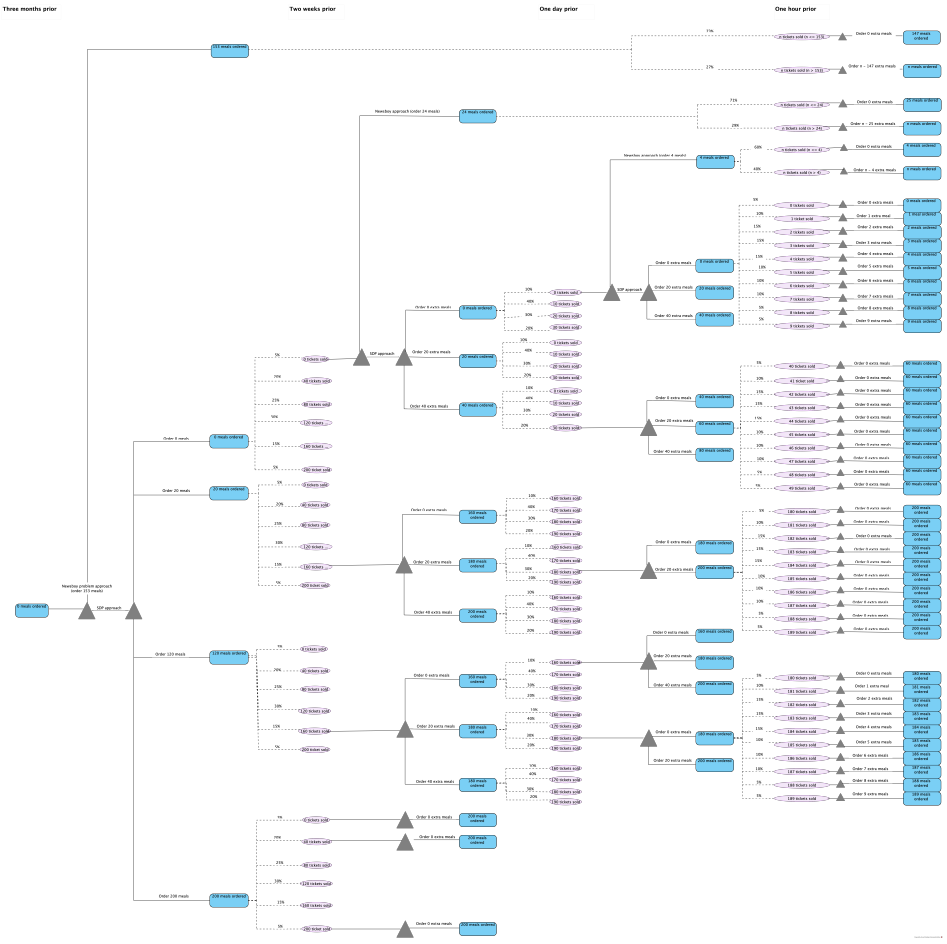

FIGURE 4: Structure of the catering problem.

We additionally made smaller diagrams of the individual decision moments to make clear what is happening. We will work backwards, so we will look at the last decision moment first. Figure 5 shows the decisions made at one day and one hour before the flight. One day prior to the flight, we can choose to order 0, 20 or 40 meals extra, while there is no decision to be made at one hour prior to the flight, since at that moment we know exactly how many tickets are sold. The costs that are in the decision boxes are the expected costs, which is the sum of the costs that are made by ordering the number of meals that are specified by the decision (immediate costs), and the products of the probability of something happening and the costs when that happens. For example, when we choose to order 0 meals at one day before the flight, we can calculate the expected costs as follows:

$$E[c] = 0 \cdot 75 + p(0) \cdot 0 + \sum_{i=1}^{9} p(i)(200 + 100i) = 600, \tag{3}$$

which can be found in the figure as well. The costs shown at the triangle is the minimum of all expected costs corresponding to the decisions that can be made. The Newsboy solution is also shown, including its expected costs. We see that ordering just 4 meals is the best decision, but this is only possible if we can order single meals. If we can only order in batches of 20, the best option is to order 0 meals at one day before the flight.

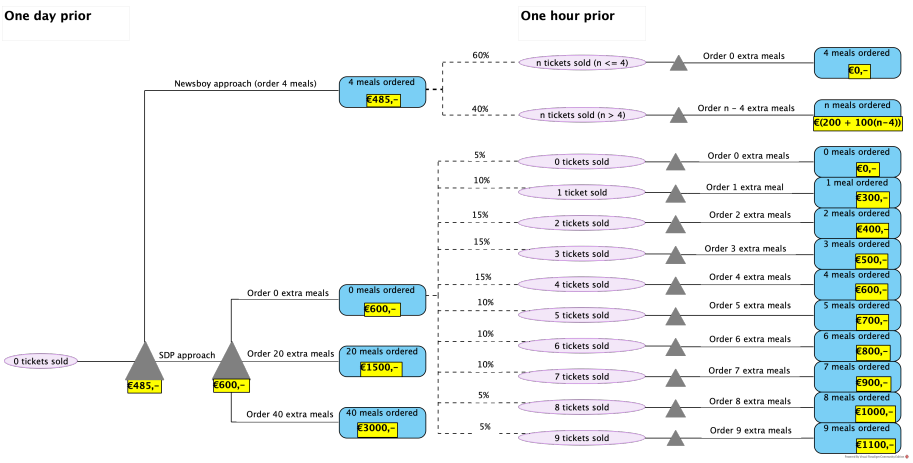

FIGURE 5: Structure of the catering problem during third and final decision moment.

Figure 6 shows the decisions that can be made two weeks prior to the flight. Again, the Newsboy solution is shown, including its expected costs. We see that ordering 24 meals is the best decision, but this is only possible if we can order single meals. If we can only order in batches of 20, the best option is to order 20 meals at two weeks before the flight. The expected costs are calculated in the same way as before.

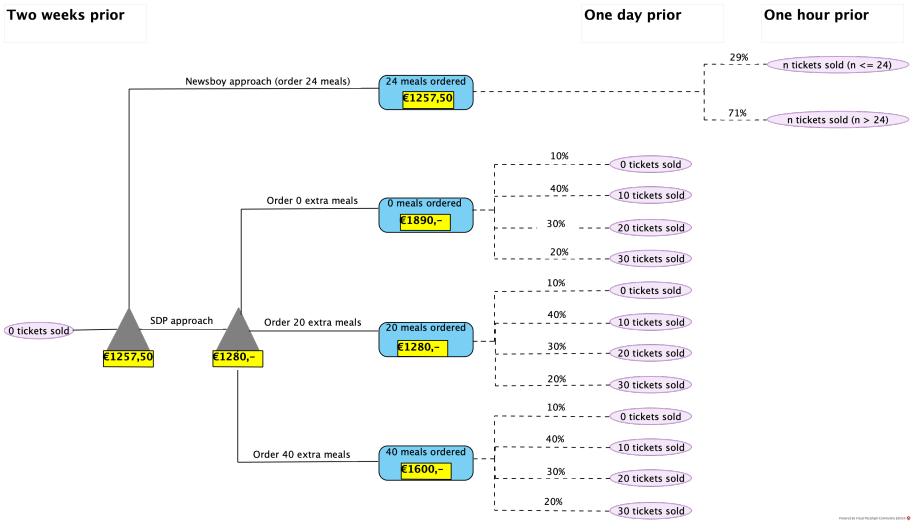

FIGURE 6: Structure of the catering problem during second decision moment.

Figure 7 shows the structure of the problem during the decision three months prior to the flight. At this moment, we can either choose to follow the Newsboy approach (Section 2.2) in which we order 153 meals immediately and order extra one hour before the flight if that is necessary, or we can choose to follow the SDP approach. We see that there are many options when choosing the SDP approach, and we see that the optimal expected costs of the SDP are smaller than the optimal expected costs for the Newsboy approach.

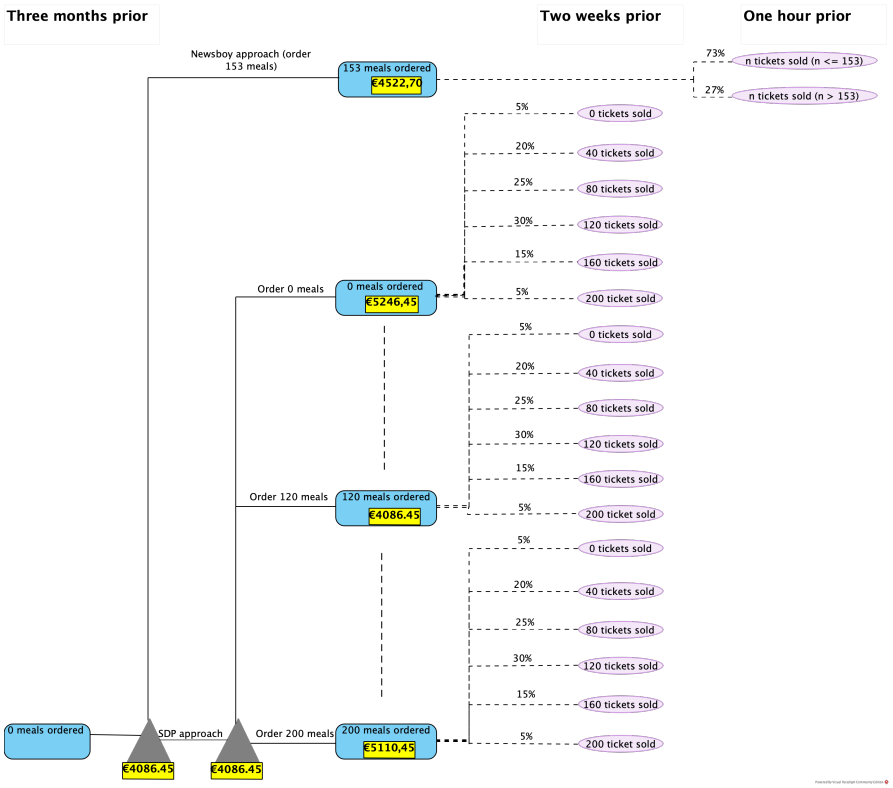

FIGURE 7: Structure of the catering problem during first decision moment.

Now, let us look at an example of possible paths. In this example, we will look at two paths that can be taken, in which at any moment the number of tickets sold is equal, but the number of meals that is ordered is different. One path will make random decisions, while the other path will take optimal decisions, according to the solution of the SDP that we will solve at the end of this section.

Figure 8 shows an example of a path that is not optimal. The red boxes show the expected costs from that moment until the end when making that decision. It follows that when following making these decisions, we eventually have to pay €8400,-, while this could have been lower as we will see in the alternative path.

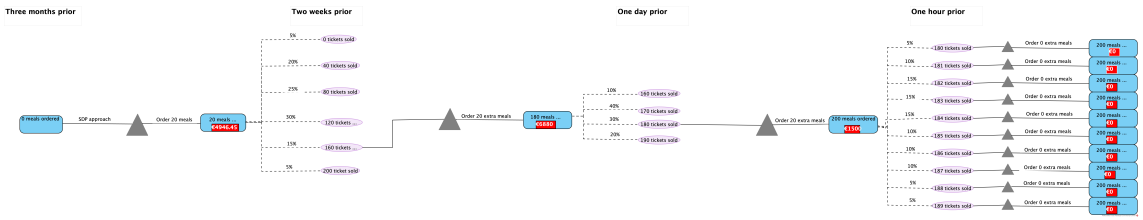

FIGURE 8: Example of a bad path.

Figure 9 shows the same path, but we will make the optimal decisions that we will find in Section 3.2 using an SDP to solve the problem. The total costs when following this path are €5400,- to €6500,-, depending on the final number of tickets sold in the last day. This is significantly lower than when following the sub optimal path from Figure 8, while the same number of tickets are sold at any moment.

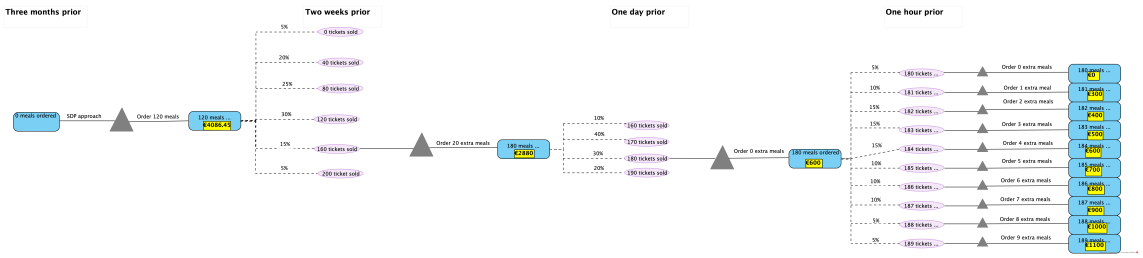

FIGURE 9: Example of an optimal path.

Now that we see the importance of making the good decisions, we will carry on with solving the problem using Stochastic Dynamic Programming.

3.2 Stochastic Dynamic Programming

We will start by setting up the Stochastic Dynamic Programming (SDP) system. A SDP system is typically described by a 5-tuple (T, S, D_t, p_t, c_t) ([4]). We will define the time T, state space S, the decision set D_t, the transition rates p_t and the costs c_t.

Time: Meals can be ordered at four different moments, namely three months, two weeks, one day and one hour before the flight. We have one extra decision moment, namely the moment where the decision to follow the SDP approach or the Newsboy approach is made. $T = \{0, 1, 2, 3, 4\}$ is the set of time, where $t = 0$ represents the moment at which can be decided to use the Newsboy approach or the SDP approach, $t = 1$ is three months before the flight, $t = 2$ is two weeks before the flight, $t = 3$ is one day before the flight and $t = 4$ is one hour before the flight.

State space: First, we will describe the state space S. This is a special Markov Decision Problem, since we need two elements to describe the state of the system, namely the number of tickets that are sold and the number of meals that are ordered up until now. Therefore, the state space is defined as $S = \{(i, j) | i \in \{0, 1, \ldots, 200\}, j \in \{0, 20, \ldots, 200\}\}$, where i is the number of tickets sold, and j is the number of meals ordered. Since the number of meals can be ordered in multiples of 20, the second element is in steps of 20.

Decision set: Furthermore, we will define the decision set D_t. The decision made at moment t depends on how many tickets are sold and the number of meals already ordered. The number of meals to order in the decision set cannot be less than the number of meals already ordered. Moreover, the number of meals can be ordered in multiples of 20, so we need to take that into account as well. Finally, it is not useful to order more than 200 meals, since that is the maximum load of the plane. The decision set is defined as $D_t(i, j) = \{j, j + 20, \ldots, 200\}$, where $j \in \mathbb{N}$ is a multiple of 20.

Transition probabilities: The transition probabilities p_{t+1} are the probabilities of a state occurring at time $t + 1$ given the previous state and the decision made at time t. They can be defined as $p((i_{t+1}, j_{t+1})|(i_t, j_t))$, where i_t represents the number of tickets sold at moment t and j_t is the number of meals ordered at moment t.

Costs: Finally, we will define the costs c_t. These costs represent the immediate costs when being in state (i, j) and making decision d at time t. As the time t grows, the meals get more expensive. In this case, we have

$$c_t(i, j, d) = \begin{cases} 25(d - j) & \text{if } t = 1,\ d > j \\ 40(d - j) & \text{if } t = 2,\ d > j \\ 75(d - j) & \text{if } t = 3,\ d > j \\ 100(d - j) + 200 & \text{if } t = 4,\ d > j \\ 0 & \text{else} \end{cases}$$

We are now able to formulate the SDP equation. We want to minimise the costs, therefore we will minimise over the decisions. We will start at $t = 4$ and work up to the solution at $t = 0$, where we can choose whether to use the Newsboy solution, or to use the SDP solution.

Let $f_t(i,j)$ be the minimised expected costs from stage t onward in state (i,j).
Then we start by:

$$f_4(i,j) = \min_{d \in D_t(i,j)} \Big\{ c_t(i,j,d) \Big\}.$$

And for $t = 3, 2, 1$:

$$f_t(i,j) = \min_{d \in D_t(i,j)} \Big\{ c_t(i,j,d) + \sum_{(k,l) \in S_{t+1}} p((k_{t+1}, l_{t+1}) | (k_t, l_t)) f_{t+1}(k,l) \Big\},$$

Finally, we are interested in f_0, in which we take the minimum of the SDP solution $f_1(0,0)$ and the Newsboy solution (Section 2.2).

$$f_t(i,j) = \min \begin{cases} 4522,70 & \text{Newsboy solution: Order 153 meals (see Section 2.2)} \\ f_1(0,0) & \text{SDP solution} \end{cases}$$

3.2.1 Results

When implementing this in Python, this results in the following findings. In Table 5, we can look up the demand at time t. Below the demand d, the optimal order and the expected costs from that point on when ordering the given number of meals are shown. The expected costs are calculated as in Equation 3, where it assumes that there are 0 meals ordered before time t. That is, the expected costs are the sum of the costs when ordering n meals and the expected costs from that point on.

Table 5: Optimal orders when using SDP.

t = 1	Demand	0					
	Optimal order	120					
	Expected costs [€]	4522,70					
t = 2	Demand	0	40	80	120	160	200
	Optimal order	20	60	100	140	180	200
	Expected costs [€]	1280	2880	4480	6080	7680	8000
t = 3	Demand	0	10	20	30	40	50
	Optimal order	0	20	20	40	40	60
	Expected costs [€]	600	1500	2100	3000	3600	4500
	Demand	60	70	80	90	100	110
	Optimal order	60	80	80	100	100	120
	Expected costs [€]	5100	6000	6600	7500	8100	9000
	Demand	120	130	140	150	160	170
	Optimal order	120	140	140	160	160	180
	Expected costs [€]	9600	10500	11100	12000	12600	13500
	Demand	180	190	200			
	Optimal order	180	200	200			
	Expected costs [€]	14100	15000	15000			

4 Results

Operations Research (OR) is a useful discipline that can help solve many similar problems, it has often helped airlines. Its methods has helped solve problems such as the overbooking problem, in which OR methods calculate how many additional tickets can be sold on top of the capacity of a flight, such that the plane will still be filled when passengers do not show up ([6]). A different problem in which OR plays a significant role is fleet assignment, in which the airline needs to assign the best size plane to a flight ([1]). In this article ([1]), there are more examples where OR is of importance of problem solving for airlines.

For the meal provisioning problem, we used three different methods that are used in Operations Research. We used Monte Carlo Simulation (Section 2.1), which gives the same results as using the Newsboy Problem (Section 2.2). Finally, we used Stochastic Dynamic Programming to solve the problem (Section 3.2), which has its own advantages.

When just ordering at one moment, it is advised to use the optimal orders of the Monte Carlo simulation and the Newsboy problem. These methods are used when there is one decision moment and they calculate the best number of meals very accurately. Since both methods find the same results, they are very strong. The results can be found in Table 4. Unfortunately, when there are multiple decision moments, these methods become too complicated.

When there are multiple decision moments as is specified in the original problem description (section 1.2), the SDP works best. It takes into account what might happen in the future, and based on that, it will calculate the optimal decision at each moment and in each possible situation. Furthermore, the SDP system can be extended to even more decision moments with arbitrary distributions, provided the distribution is discrete. The optimal decisions can be found in Table 5.

References

[1] Cynthia Barnhart, Peter Belobaba, and Amedeo R Odoni. Applications of Operations Research in the Air Transport Industry. *Transportation Science*, 37(4), 2003.

[2] Jason H. Goto, Mark E. Lewis, and Martin L. Puterman. Coffee, Tea, or...?: A Markov Decision Process Model for Airline Meal Provisioning. *Transportation Science*, 38(1):107–118, 2004.

[3] Moutaz Khouja. The single-period (news-vendor) problem: Literature review and suggestions for future research. *Omega*, 27(5):537–553, 10 1999.

[4] Mihaela Mitici. MDP for Query-Based Wireless Sensor Networks. In R. J. Boucherie and N.M. van Dijk, editors, *Markov Decision Processes in Practice*, chapter 20, pages 505–519. Sprinter International Publishing, Cham, 2017.

[5] Martin L. Puterman. Markov decision processes: Discrete stochastic dynamic programming. In *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, pages 1–649. wiley, 1 2008.

[6] Marvin Rothstein. OR AND THE AIRLINE OVERBOOKING PROBLEM. *Operations Research*, 33(2):237–248, 1985.

Appendices

A Monte Carlo Simulation Code

```
import numpy as np
import scipy.stats as st
import pandas as pd
import matplotlib.pyplot as plt

k = 10000000
l = 201
fine = 200
unit1 = 25
unit2 = 40
unit3 = 100

mu = 150
sd = 30
data = pd.ExcelFile(r'data.xlsx')


def get_probability(index, number):
    sheets = data.sheet_names
    temp = data.parse(sheets[index - 1]).Demand
    for i in range(len(temp)):
        if temp[i] == number:
            return data.parse(sheets[index - 1]).Probability[i]


def total_prob(number):
    if number < 0 or number > 240:
        return 0
    else:
        week2 = (number // 40) * 40
        week3 = ((number - week2) // 10) * 10
        week4 = number - week2 - week3
        return get_probability(1, week2) * get_probability(2,
                               week3) * get_probability(3, week4)


def costs(demand, order):
    demand = min(200, demand)
    if order > demand:  # too many ordered
        result = unit1 * order
    elif demand >= order:  # too few ordered
        result = unit1 * order + (demand - order) * unit3 + fine
    return result


def determine_costs(index):
    test_costs = np.empty((l, k))
    values = np.array(range(240))
    probabilities = np.empty(240)
    for i in range(240):
```

```
        probabilities[i] = total_prob(i)

    for i in range(0, l):
        if index == 1:  # standard normal with 150, 30
            test = np.random.normal(mu, sd, k)
        elif index == 2:  # according to all combined distributions
            test = np.random.choice(a=values, size=k,
                                    p=probabilities)
        for j in range(0, k):
            test_costs[i][j] = costs(test[j], i)
    return test_costs


def get_ci(test_costs):
    conf_interval = np.empty((l, 2))
    for i in range(0, l):
        stdev = st.tstd(test_costs[i, :])
        conf_interval[i, 0] = np.mean(test_costs[i, :]) - \
                              st.norm.ppf(.975) * stdev / k
        conf_interval[i, 1] = np.mean(test_costs[i, :]) + \
                              st.norm.ppf(.975) * stdev / k
    return conf_interval


def get_optimum(test_costs):
    mean = np.mean(test_costs, axis=1)
    optimum = np.argmin(mean)
    return optimum


def to_string(test_costs):
    optimum = get_optimum(test_costs)
    CI = get_ci(test_costs)[optimum]

    print("At this point, it is best to order a total of",
          optimum, "meals.")
    print("The 95% interval of the costs when ordering this "
          "many meals is", CI)


def graph(test_costs):
    optimum = get_optimum(test_costs)
    y = np.mean(test_costs, axis=1)[optimum-5:optimum+6]
    x = list(range(optimum-5, optimum+6))

    plt.plot(x, y, label="costs CI")
    plt.xticks(x)
    plt.legend()
    plt.show()


costs = determine_costs(2)
to_string(costs)
graph(costs)
```

B Newsboy Code

```
import numpy as np
import scipy.stats as st
import pandas as pd


def get_demand(index):
    if index == 1:
        return [0]
    elif index == 2:
        return range(0, 201, 40)
    elif index == 3:
        return range(0, 201, 10)
    return 0


class Newsboy:

    def __init__(self):
        self.data = pd.ExcelFile(r'data.xlsx')

    def get_probability(self, index, number):
        sheets = self.data.sheet_names
        temp = self.data.parse(sheets[index - 1]).Demand
        for i in range(len(temp)):
            if temp[i] == number:
                return self.data.parse(sheets[index -
                                               1]).Probability[i]

    def total_prob1(self, number):
        if number < 0 or number > 240:
            return 0
        else:
            week2 = (number // 40) * 40
            week3 = ((number - week2) // 10) * 10
            week4 = number - week2 - week3
            return self.get_probability(1, week2) * \
                   self.get_probability(2, week3) * \
                   self.get_probability(3, week4)

    def total_prob2(self, number, current):
        if number < current or number > current + 39:
            return 0
        else:
            extra = number - current
            week3 = (extra // 10) * 10
            week4 = extra - week3
            return self.get_probability(2, week3) * \
                   self.get_probability(3, week4)

    def total_prob3(self, number, current):
        if number < current or number > current + 9:
            return 0
        else:
            extra = number - current
```

```
55              return self.get_probability(3, extra)
56
57      def total_prob(self, number, index, current=0):
58          if index == 1:
59              return self.total_prob1(number)
60          elif index == 2:
61              return self.total_prob2(number, current)
62          elif index == 3:
63              return self.total_prob3(number, current)
64
65      def pdf(self, index, current=0):
66          result = np.empty(240)
67          for i in range(240):
68              result[i] = self.total_prob(i, index, current)
69          return result
70
71      def cdf_recursive(self, index, current=0):
72          result = np.empty(240)
73          result[0] = self.total_prob(0, index, current)
74          for i in range(1, 240):
75              result[i] = result[i - 1] + \
76                          self.total_prob(i, index, current)
77
78          return result
79
80      def cost(self, index):
81          if index == 4:
82              index = 1
83          temp = self.data.parse('Costs').Costs
84          return temp[index - 1]
85
86      def f(self, order, index, costs, cdf=None, pdf=None):
87          if index == 4:
88              return costs + 100 * (st.norm.cdf(order, 150, 30) -
89                      2 * st.norm.pdf(order, 150, 30) - 1)
90          else:
91              return costs * (1 - cdf[0]) + \
92                     100 * (cdf[order] - 2 * pdf[order] - 1)
93
94      def find_optimum(self, index, current=0):
95          costs = self.cost(index)
96          temp = np.empty(201 - current)
97          if index == 4:
98              for j in range(current, 201):
99                  temp[j - current] = self.f(j, index, costs)
100             optimum = np.argmin(abs(temp))
101             return optimum + current
102         else:
103             cdf = self.cdf_recursive(index, current)
104             cdf[200] = cdf[239]
105             cdf = cdf[0:201]
106             pdf = self.pdf(index, current)
107             for i in range(201, 240):
108                 pdf[200] = pdf[200] + pdf[i]
109             pdf = pdf[0:201]
110
```

```
            for j in range(current, 201):
                temp[j - current] = self.f(j, index, costs,
                                           cdf, pdf)
            optimum = np.argmin(abs(temp))
            return optimum + current

    def expected_costs(self, index, order, current):
        pdf = self.pdf(index, current)
        costs = self.cost(index)

        extra_costs = np.zeros(240)
        for i in range(240):
            if i > order and order < 200:
                extra_costs[i] = pdf[i] * \
                                 (200 + (min(i, 200) - order)
                                  * 100)
        return sum(extra_costs) + costs * order

    def all_choices(self, index):
        data = get_demand(index)
        result = np.vstack([data, np.empty([2, len(data)])])

        for i in range(len(data)):
            choice = self.find_optimum(index, data[i])
            result[1, i] = choice
            result[2, i] = self.expected_costs(index,
                                               choice, data[i])
        df = pd.DataFrame(data=result, index=["current demand",
                                              "optimal order",
                                              "expected costs"],
                          columns=[""] * len(data))
        return df

    def to_string(self):
        with pd.option_context('display.max_rows', None,
                               'display.max_columns', None):
            for i in range(3):
                print("Stage {}:".format(i + 1),
                      self.all_choices(i + 1), "\n")


newsboy = Newsboy()
newsboy.to_string()
```

C Stochastic Dynamic Programming Code

```
import numpy as np
import pandas as pd


def get_boolean(string):
    while True:
        answer = input(string)
        if answer == 'y':
            return True
        elif answer == 'n':
            return False
        else:
            print("Oops! You need to enter y or n. ")


class SDP:

    def __init__(self, costs, stage=None, current_order=None,
                 current_demand=None):
        self.data = pd.ExcelFile(r'data.xlsx')
        self.total_seats = 200
        self.order_step = 20
        self.total_moments = len(self.data.sheet_names)
        self.results = np.full(self.total_moments, -1,
                               dtype=object)

        if stage is None:
            self.get_stage()
        else:
            self.stage = stage

        if current_order is None:
            self.get_ordered()
        else:
            self.current_order = current_order

        if current_demand is None:
            self.get_demand()
        else:
            self.current_demand = current_demand

        self.costs = 0
        if costs:
            self.get_costs()
        else:
            self.get_price()

    def get_price(self):
        temp1 = get_boolean("Do you want to use the default "
                            "values for the costs? (y/n) ")
        if temp1:
            self.get_costs()
        if not temp1:
            self.costs = np.empty(self.total_moments + 1)
```

```
            for i in range(4 - self.stage + 2):
                if self.stage > 1:
                    self.costs[0:(self.stage - 1)] = 0
                while True:
                    try:
                        if i == 4 - self.stage + 1:
                            result = input("Cost of the fine for "
                                           "having too few "
                                           "meals: ")
                            self.costs[i + self.stage - 1] \
                                = result
                            break
                        result = input("Cost of one meal at stage"
                                       " {}: ".format(i +
                                                      self.stage))
                        result = int(result)
                        self.costs[i + self.stage - 1] = result
                        break
                    except ValueError:
                        print("Oops! You need to enter an "
                              "integer!")

    def get_costs(self):
        temp = self.data.parse('Costs').Costs
        costs = np.empty(len(temp))
        for i in range(len(costs)):
            costs[i] = temp[i]
        self.costs = costs

    def get_stage(self):
        while True:
            try:
                stage = input("1: 3 months before flight \n"
                              "2: 2 weeks before flight \n"
                              "3: 1 day before flight \n"
                              "At which stage are you? ")
                stage = int(stage)
                if stage < 1 or stage > 3:
                    raise ValueError
                else:
                    self.stage = stage
                break
            except ValueError:
                print("Oops! You need to enter an integer between"
                      " 1 and 3! Let's try again. \n")
        return stage

    def get_ordered(self):
        while True:
            try:
                if self.stage == 1:
                    self.current_order = 0
                    break
                current_order = input("How many meals are ordered"
                                      " at this point? ")
                self.current_order = int(current_order)
```

```
                if self.current_order % 20 != 0 or \
                        self.current_order < 0 or \
                        self.current_order > self.total_seats:
                    raise ValueError
                break
            except ValueError:
                print("Oops! You need to enter an integer between"
                      " 0 and" + str(self.total_seats) +
                      ", which is a multiple of 20! Let's "
                      "try again. \n")

    def get_demand(self):
        while True:
            try:
                if self.stage == 1:
                    self.current_demand = 0
                    break
                current_demand = input("How many tickets are sold"
                                       " at this moment? ")
                self.current_demand = int(current_demand)
                if self.current_demand % 40 != 0 or \
                        self.current_demand < 0 or \
                        self.current_demand > self.total_seats:
                    if self.stage == 2:
                        raise ValueError
                    elif self.stage == 3:
                        if self.current_demand % 10 == 0:
                            if self.current_demand > \
                                    self.current_order \
                                    and self.current_demand - \
                                    self.current_order != \
                                    (0 or 10 or 20 or 30):
                                raise ValueError
                        else:
                            raise ValueError
                    if self.stage != 2 and \
                            self.current_demand % 10 == 0:
                        break
                    break
                break
            except ValueError:
                if self.stage == 2:
                    print("Oops! You need to enter an integer"
                          " between 0 and 200, which is a "
                          "multiple of 40! Let's try again. ")
                if self.stage == 3:
                    print("Oops! You need to enter an integer "
                          "between 0 and 200, which is a "
                          "multiple of 10! The demand should "
                          "be smaller than the number of meals "
                          "ordered, or it should be",
                          self.current_order, "+ 0, + 10, + 20 "
                                              "or + 30. Let's "
                                              "try again. ")
                else:
                    print("Oops! You need to enter an integer "
```

```
                            "between 0 and 200, which is a "
                            "multiple of 10! Let's try again. ")

    def expected_costs(self, index, ordered, next_order):
        if index != self.total_moments:
            return max(0, (next_order - ordered)
                       * self.costs[index - 1])
        elif index == self.total_moments:
            if next_order > ordered:
                return self.costs[index] + (next_order - ordered)\
                       * self.costs[index - 1]
            else:
                return 0

    def sdp(self, index, ordered, demand):
        if demand > 200:
            demand = 200
        if index != self.total_moments:
            next_order = np.arange(0, self.total_seats + 1, 20)
            next_order = next_order[next_order >= ordered]
            reward = np.empty(len(next_order))

            next_demand = self.get_next_demand(index, demand)
            prob = self.get_probability(index)
            fnext = np.empty((len(next_order), len(next_demand)))

            for i in range(len(next_order)):
                if next_order[i] >= ordered:
                    reward[i] = self.expected_costs(index,
                                                    ordered,
                                                    next_order[i])
                    for j in range(len(next_demand)):
                        fnext[i, j] = min(self.sdp(index + 1,
                                                   next_order[i],
                                                   next_demand[j]))
                else:
                    reward[i] = np.inf
                    fnext[i, :] = np.inf

            product = prob * fnext
            result = np.empty(len(next_order))
            for i in range(len(next_order)):
                temp = sum(product[i, :])
                result[i] = reward[i] + temp
        else:
            result = [self.expected_costs(index, ordered, demand)]
        return result

    def get_probability(self, index):
        sheets = self.data.sheet_names
        temp = self.data.parse(sheets[index - 1]).Probability
        result = np.empty(len(temp))
        for i in range(len(temp)):
            result[i] = temp[i]
        return result

```

```
223     def get_next_demand(self, index, demand):
224         sheets = self.data.sheet_names
225         temp = self.data.parse(sheets[index - 1]).Demand
226         result = np.empty(len(temp))
227         for i in range(len(temp)):
228             result[i] = temp[i]
229         return result + demand
230 
231     def best_choice(self, index, ordered, demand):
232         temp = self.sdp(index, ordered, demand)
233         best_order = np.argmin(temp)
234         next_order = np.arange(0, self.total_seats + 1, 20)
235         next_order = next_order[next_order >= ordered]
236         result = next_order[best_order]
237         return [result, temp[best_order]]
238 
239     def all_choices(self, index):
240         data = self.get_data(index, np.zeros(1))
241         result = np.vstack([data, np.empty([2, len(data)])])
242 
243         for i in range(len(data)):
244             choice = self.best_choice(index, 0, data[i])
245             result[1, i] = choice[0]
246             result[2, i] = choice[1]
247         df = pd.DataFrame(data=result, index=["current demand",
248                                               "optimal order",
249                                               "expected costs"],
250                           columns=[""] * len(data))
251         return df
252 
253     def get_data(self, index, demand):
254         if index > 1:
255             sheets = self.data.sheet_names
256             data = self.data.parse(sheets[index - 2]).Demand
257             new_demand = np.zeros(len(demand) * len(data))
258             k = 0
259             for i in range(len(data)):
260                 for j in range(len(demand)):
261                     new_demand[k] = demand[j] + data[i]
262                     k += 1
263             result = self.get_data(index - 1, new_demand)
264             return result
265         else:
266             result = demand
267             return result[result <= self.total_seats]
268 
269     def to_string(self):
270         with pd.option_context('display.max_rows', None,
271                                'display.max_columns', None):
272             for i in range(self.total_moments - 1):
273                 print("Stage {}: ".format(i + 1) +
274                       self.all_choices(i + 1).__str__() + "\n\n")
275 
276     def main(self):
277         optimum = self.best_choice(self.stage, self.current_order,
278                                    self.current_demand)
```

```
279         print("The best choice is to order a total of",
280               optimum[0], "meals at this point. \n This will "
281                           "approximately cost", optimum[1],
282               "Euro including this order until the flight")
283
284
285 stage = 1
286 sdp = SDP(True, stage)
287 sdp.to_string()
```

D Probability Distribution Function

TABLE 6: Complete distribution of the final number of tickets sold when having no information at three months before the flight.

Demand	Probability
0	0.00025
1	0.0005
2	0.00075
3	0.00075
4	0.00075
5	0.0005
6	0.0005
7	0.0005
8	0.00025
9	0.00025
10	0.001
11	0.002
12	0.003
13	0.003
14	0.003
15	0.002
16	0.002
17	0.002
18	0.001
19	0.001
20	0.00075
21	0.0015
22	0.00225
23	0.00225
24	0.00225
25	0.0015
26	0.0015
27	0.0015
28	0.00075
29	0.00075
30	0.0005
31	0.001
32	0.0015
33	0.0015
34	0.0015
35	0.001
36	0.001
37	0.001
38	0.0005
39	0.0005
40	0.001
41	0.002
42	0.003
43	0.003
44	0.003
45	0.002
46	0.002
47	0.002
48	0.001
49	0.001
50	0.004
51	0.008
52	0.012
53	0.012
54	0.012
55	0.008
56	0.008
57	0.008
58	0.004
59	0.004
60	0.003
61	0.006
62	0.009
63	0.009

Demand	Probability
64	0.009
65	0.006
66	0.006
67	0.006
68	0.003
69	0.003
70	0.002
71	0.004
72	0.006
73	0.006
74	0.006
75	0.004
76	0.004
77	0.004
78	0.002
79	0.002
80	0.00125
81	0.0025
82	0.00375
83	0.00375
84	0.00375
85	0.0025
86	0.0025
87	0.0025
88	0.00125
89	0.00125
90	0.005
91	0.01
92	0.015
93	0.015
94	0.015
95	0.01
96	0.01
97	0.01
98	0.005

Demand	Probability
99	0.005
100	0.00375
101	0.0075
102	0.01125
103	0.01125
104	0.01125
105	0.0075
106	0.0075
107	0.0075
108	0.00375
109	0.00375
110	0.0025
111	0.005
112	0.0075
113	0.0075
114	0.0075
115	0.005
116	0.005
117	0.005
118	0.0025
119	0.0025
120	0.0015
121	0.003
122	0.0045
123	0.0045
124	0.0045
125	0.003
126	0.003
127	0.003
128	0.0015
129	0.0015
130	0.006
131	0.012
132	0.018
133	0.018

Demand	Probability
134	0.018
135	0.012
136	0.012
137	0.012
138	0.006
139	0.006
140	0.0045
141	0.009
142	0.0135
143	0.0135
144	0.0135
145	0.009
146	0.009
147	0.009
148	0.0045
149	0.0045
150	0.003
151	0.006
152	0.009
153	0.009
154	0.009
155	0.006
156	0.006
157	0.006
158	0.003
159	0.003
160	0.00075
161	0.0015
162	0.00225
163	0.00225
164	0.00225
165	0.0015
166	0.0015
167	0.0015
168	0.00075

Demand	Probability
169	0.00075
170	0.003
171	0.006
172	0.009
173	0.009
174	0.009
175	0.006
176	0.006
177	0.006
178	0.003
179	0.003
180	0.00225
181	0.0045
182	0.00675
183	0.00675
184	0.00675
185	0.0045
186	0.0045
187	0.0045
188	0.00225
189	0.00225
190	0.0015
191	0.003
192	0.0045
193	0.0045
194	0.0045
195	0.003
196	0.003
197	0.003
198	0.0015
199	0.0015
200	0.00025
201	0.0005
202	0.00075
203	0.00075

Demand	Probability
204	0.00075
205	0.0005
206	0.0005
207	0.0005
208	0.00025
209	0.00025
210	0.001
211	0.002
212	0.003
213	0.003
214	0.003
215	0.002
216	0.002
217	0.002
218	0.001
219	0.001
220	0.00075
221	0.0015
222	0.00225
223	0.00225
224	0.00225
225	0.0015
226	0.0015
227	0.0015
228	0.00075
229	0.00075
230	0.0005
231	0.001
232	0.0015
233	0.0015
234	0.0015
235	0.001
236	0.001
237	0.001
238	0.0005

Demand	Probability
239	0.0005

TABLE 7: Distribution two weeks before the flight when the current demand is 0.

Demand	**Probability**
0	0.005
1	0.01
2	0.015
3	0.015
4	0.015
5	0.01
6	0.01
7	0.01
8	0.005
9	0.005
10	0.02
11	0.04
12	0.06
13	0.06
14	0.06
15	0.04
16	0.04
17	0.04
18	0.02
19	0.02
20	0.015
21	0.03
22	0.045
23	0.045
24	0.045
25	0.03
26	0.03
27	0.03
28	0.015
29	0.015
30	0.01
31	0.02
32	0.03
33	0.03
34	0.03
35	0.02
36	0.02
37	0.02
38	0.01
39	0.01

TABLE 8: Distribution two weeks before the flight when the current demand is 40.

Demand	**Probability**
40	0.005
41	0.01
42	0.015
43	0.015
44	0.015
45	0.01
46	0.01
47	0.01
48	0.005
49	0.005
50	0.02
51	0.04
52	0.06
53	0.06
54	0.06
55	0.04
56	0.04
57	0.04
58	0.02
59	0.02
60	0.015
61	0.03
62	0.045
63	0.045
64	0.045
65	0.03
66	0.03
67	0.03
68	0.015
69	0.015
70	0.01
71	0.02
72	0.03
73	0.03
74	0.03
75	0.02
76	0.02
77	0.02
78	0.01
79	0.01

TABLE 9: Distribution two weeks before the flight when the current demand is 80.

Demand	**Probability**
80	0.005
81	0.01
82	0.015
83	0.015
84	0.015
85	0.01
86	0.01
87	0.01
88	0.005
89	0.005
90	0.02
91	0.04
92	0.06
93	0.06
94	0.06
95	0.04
96	0.04
97	0.04
98	0.02
99	0.02
100	0.015
101	0.03
102	0.045
103	0.045
104	0.045
105	0.03
106	0.03
107	0.03
108	0.015
109	0.015
110	0.01
111	0.02
112	0.03
113	0.03
114	0.03
115	0.02
116	0.02
117	0.02
118	0.01
119	0.01

TABLE 10: Distribution two weeks before the flight when the current demand is 120.

Demand	**Probability**
120	0.005
121	0.01
122	0.015
123	0.015
124	0.015
125	0.01
126	0.01
127	0.01
128	0.005
129	0.005
130	0.02
131	0.04
132	0.06
133	0.06
134	0.06
135	0.04
136	0.04
137	0.04
138	0.02
139	0.02
140	0.015
141	0.03
142	0.045
143	0.045
144	0.045
145	0.03
146	0.03
147	0.03
148	0.015
149	0.015
150	0.01
151	0.02
152	0.03
153	0.03
154	0.03
155	0.02
156	0.02
157	0.02
158	0.01
159	0.01

TABLE 11: Distribution two weeks before the flight when the current demand is 160.

Demand	**Probability**
160	0.005
161	0.01
162	0.015
163	0.015
164	0.015
165	0.01
166	0.01
167	0.01
168	0.005
169	0.005
170	0.02
171	0.04
172	0.06
173	0.06
174	0.06
175	0.04
176	0.04
177	0.04
178	0.02
179	0.02
180	0.015
181	0.03
182	0.045
183	0.045
184	0.045
185	0.03
186	0.03
187	0.03
188	0.015
189	0.015
190	0.01
191	0.02
192	0.03
193	0.03
194	0.03
195	0.02
196	0.02
197	0.02
198	0.01
199	0.01

TABLE 12: Distribution two weeks before the flight when the current demand is 200.

Demand	**Probability**
200	1