Chair of Information Technology

Master's Thesis

Online Anticipatory Algorithms for Scheduling Problems

Simon Erler, BSc

February 2021

Abstract

This work considers an online packet scheduling problem where packets arrive independently over a discrete time horizon and the goal is to minimize the cumulative weighted packet loss. The significant challenge of this problem is that the arrival model is not known in advance and may underlie dynamic changes. An important practical application of this setting is the scheduling of arriving IP packets in computer networks.

The focus lies on the definition of *online anticipatory algorithms* that achieve an improvement over the oblivious approach of the greedy algorithm when scheduling requests in an uncertain, dynamic environment. The concept of *anticipation* is developed in this context by incorporating information of the environment's history to predict certain aspects of the future.

Two distinct approaches are presented within the scope of this work: *reinforcement learning* and *online stochastic combinatorial optimization*. The theoretical background of both concepts is discussed in detail and the performance of the developed algorithms is analysed on the online packet scheduling problem.

The experimental analysis shows that online stochastic combinatorial optimization yields the smallest cumulative weighted loss in any setting if the input distribution is modelled by Markov chains. However, it also requires the significantly largest runtime for each decision. To cope with a non-Markovian environment, first a conservative approach for the Q-learning algorithm is proposed that compared to the greedy algorithm achieves a significant improvement for the 2-class and 3-class problem. When more packet classes are present, the classical Q-learning algorithm has been found to be the best approach. However, it was not able to outperform greedy for the n-packet problem within the simulated time horizon, for $n \geq 4$.

Kurzfassung

Diese Arbeit befasst sich mit einer Variante des *Online-Packet-Scheduling*-Problems, wobei Pakete unabhängig voneinander über einen diskreten Zeithorizont eintreffen und das Ziel in der Minimierung des kumulierten gewichteten Paketverlustes liegt. Die Herausforderung des Problems besteht hauptsächlich darin, dass der Ankunftsprozess nicht bekannt ist und dynamischen Veränderungen unterliegen kann. Eine wichtige praktische Anwendung ist die Allokation von eintreffenden IP-Paketen in Computernetzwerken.

Der Fokus liegt in der Untersuchung von *Online-Anticipatory*-Algorithmen, die im Vergleich zum Greedy Algorithmus eine Verbesserung der Allokation in einer unbekannten, dynamischen Umgebung erreichen. Beobachtungen aus der Vergangenheit werden dazu verwendet, Prognosen für die Zukunft zu erstellen, um ein vorausschauendes Handeln zu ermöglichen.

Im Rahmen der Arbeit werden zwei Ansätze vorgestellt: *Reinforcement Learning* und *Online Stochastic Combinatorial Optimization*. Der theoretische Hintergrund beider Konzepte wird genau erklärt und die Performance der entwickelten Algorithmen wird anhand des *Online-Packet-Scheduling*-Problems analysiert.

Die durchgeführten Experimente zeigen, dass *Online Stochastic Combinatorial Optimization* den geringsten gewichteten kumulierten Paketverlust liefert, wenn der Ankunftsprozess durch Markov-Modelle beschrieben wird. Allerdings benötigt dies auch die signifikant größte Laufzeit für jede Entscheidung. Für den Fall, dass die Markov-Annahme nicht gilt, wird zuerst eine konservative Q-Learning-Strategie vorgeschlagen, welche im Vergleich zu Greedy eine deutliche Verbesserung für das 2-Klassen- und 3-Klassen-Problem erreicht. Für mehr als drei Klassen ist der gewöhnliche Q-Learning-Algorithmus besser geeignet. Jedoch konnte für diesen Fall keine Verbesserung gegenüber Greedy innerhalb des simulierten Zeithorizontes erreicht werden.

Acknowledgements

I wish to thank, first and foremost, the Chair of Information Technology and my supervisor Dr. rer. nat. Ronald Ortner for his guidance and assistance.
Furthermore I want to thank Univ.-Prof. Dr. Peter Auer for his interesting lecture in Machine Learning, which was very helpful when approaching the topic of reinforcement learning.
At last I want to thank my parents for enabling my studies and for supporting me in all my decisions.

Danksagung

Ich möchte mich an dieser Stelle als erstes am Lehrstuhl für Informationstechnologie bedanken, sowie bei meinen Betreuer Dr. rer. nat. Ronald Ortner für seine Anleitung und Hilfestellung.
Weiters bedanke ich mich bei Univ.-Prof. Dr. Peter Auer für seine interessante Lehrveranstaltung zum Thema Maschinelles Lernen, welche sehr hilfreich beim Herangehen des Themas Reinforcement Learning war.
Zuletzt gilt mein besonderer Dank meinen Eltern, die mir mein Studium ermöglichen und mich in all meinen Entscheidungen unterstützt haben.

MONTANUNIVERSITÄT LEOBEN
www.unileoben.ac.at

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

Ich erkläre, dass ich die Richtlinien des Senats der Montanuniversität Leoben zu "Gute wissenschaftliche Praxis" gelesen, verstanden und befolgt habe.

Weiters erkläre ich, dass die elektronische und gedruckte Version der eingereichten wissenschaftlichen Abschlussarbeit formal und inhaltlich identisch sind.

Datum 12.02.2021

Unterschrift Verfasser/in
Simon Erler

Contents

1 Introduction **1**
1.1 Online Algorithms 2
1.1.1 Competitive Ratio 3
1.1.2 Regret Minimization 4
1.2 Online Scheduling 4
1.3 Anticipation 5

2 Online Stochastic Optimization **7**
2.1 Stochastic Programming 7
2.2 Stochastic Combinatorial Optimization 8
2.2.1 Sampling Future Tasks 9
2.2.2 Anticipativity Assumption 9
2.3 Online Anticipatory Algorithms 11

3 Reinforcement Learning **12**
3.1 Finite Markov Decision Processes 13
3.1.1 Reward Signal and Return 14
3.1.2 Optimal Value Function 14
3.2 Q-Learning 15
3.3 Exploration and Exploitation 16
3.3.1 Epsilon Greedy 17
3.3.2 UCB-1 17
3.4 Non-Markovian Observations 18
3.4.1 Conservative Q-Learning 18
3.4.2 Belief States 19

4 Online Packet Scheduling **20**
4.1 Problem Definition 20
4.2 Offline Optimization 21
4.2.1 Postprocessing 23
4.3 Oblivious Online Packet Scheduling 24

4.3.1 Greedy Algorithm 24
4.3.2 Local Optimization 25
4.4 Anticipative Online Packet Scheduling 26
4.4.1 Expectation Algorithm 27
4.4.2 Consensus Algorithm 29
4.4.3 Regret Algorithm 30
4.4.4 Q-Learning Algorithm 33
4.4.5 Conservative Q-Learning Algorithm 34
4.4.6 Q-Learning Algorithm with Belief States 35

5 Learning Input Distributions 37
5.1 Hidden Markov Models 38
5.1.1 Forward and Backward Algorithm 39
5.1.2 Baum-Welch Algorithm 40
5.1.3 Precision Range and Scaling 43
5.2 Historical Averaging 44
5.3 Historical Sampling 45
5.4 Machine Learning 46

6 Experimental Analysis 48
6.1 Experimental Setting 49
6.2 Oblivious Algorithms 50
6.3 Stochastic Optimization Algorithms 51
6.4 Reinforcement Learning Algorithms 54

7 Conclusion 57

List of Figures

1.1 Optimal Decision Sequence with Anticipative Behaviour . . . 5

3.1 Reinforcement Learning Scenario 12

4.1 Offline Optimization Algorithm $\mathcal{O}$ 22
4.2 Postprocessing Step of the Offline Optimal Solution 23
4.3 Online Greedy Algorithm $\mathcal{G}$ 25
4.4 Online Local Optimization Algorithm $\mathcal{L}$ 25
4.5 Generic Online Algorithm $\mathcal{A}$ 26
4.6 Expectation Algorithm $\mathcal{E}$ 27
4.7 Consensus Algorithm $\mathcal{C}$ 29
4.8 Regret Algorithm $\mathcal{R}$ 30
4.9 Suboptimality Approximation Regret Calculation 32
4.10 Q-Learning Algorithm $\mathcal{RL}$ 33
4.11 Conservative Q-Learning Algorithm $\mathcal{RLC}$ 35

5.1 Generic Online Algorithm $\mathcal{A}'$ with Learning 38
5.2 Algorithm for Learning Hidden Markov Models 42
5.3 Implementation of Historical Averaging 45
5.4 Implementation of Historical Sampling 46

6.1 Experimental Model of the Packet Arrival 49
6.2 Comparison of Greedy $\mathcal{G}$ and Local Optimization $\mathcal{L}$ 50
6.3 Effect of the Postprocessing step in the Online Framework . . 51
6.4 Comparison of the Stochastic Optimization Algorithms 52
6.5 Sampling Methods for the Stochastic Optimization Algorithms 53
6.6 Initial Exploration Loss on the 3-Class Problem 54
6.7 Comparison of the Reinforcement Learning Algorithms 55

Chapter 1

Introduction

Traditional optimization systems generally have focused on a priori optimization and have therefore not been able to react to disturbances or unexpected events. The progress in optimization over the last decades however enables advanced optimization techniques that collect data in real time and adaptively improve their decisions [1]. *Online algorithms* represent a theoretical framework for studying problems where the input in an interactive system arrives as a sequence of input fragments and the system has to react responding to each incoming fragment, considering that all future fragments are not known. Over the last years online algorithms have received substantial attention and have been studied in many application areas, such as resource management, data structuring, scheduling, or finance [2].

The process within the online framework, consisting in chronological decisions for a dynamic problem, is also called *dynamic decision process* [3]. Multistage stochastic programs can be used to describe such a scenario, but finding optimal policies for large-scale multistage stochastic optimization problems is not feasible using existing methods [4]. Many large real-world problems however are in fact dynamic, thus they change over time, whereby the changes are generally not known beforehand. Therefore, since a priori optimization cannot handle such unexpected changes, these problems must be solved online.

In the online framework, at each time step a single decision is chosen based on the current knowledge of the system and its environment. To further improve a decision beyond this oblivious approach, the notion of *anticipation* has been developed in this context. Anticipation has been largely discussed in science and been given various definitions and interpretations. A possible way of defining anticipation is by a system that contains a predictive model of itself and/or its environment [5]. Also, the term *anticipative behaviour*, which will be frequently used, is closely related to the notion of anticipation.

Anticipative behaviour means that decisions do not only depend on the past and present, but also on predictions, expectations or beliefs about the future [6]. In this work, anticipation is achieved by either predicting future possible requests or by gathering experience from direct interaction with the environment in the past. Technically, also predicting the future in general relies on information learned throughout the history, so anticipation is strongly related to learning from the experienced past. A learning system improves its performance through experience gained over a period of time without complete information about the environment in which it operates [7].

This work aims to define *online anticipatory algorithms* that can be applied to scheduling problems. Online anticipatory algorithms are algorithms that sustain the online framework and select their decisions such that anticipative behaviour is incorporated. Scheduling problems are interesting in this context, since they often arise in the online framework naturally. The main issues in online scheduling are the management of uncertainty and time restrictions of decision-making [8]. Incoming tasks, requests, or jobs are usually not known beforehand and therefore a priori optimization is not feasible. Furthermore, some structure and time dependency is expected in the arrival sequence of most scheduling problems, which is a fundamental requirement for obtaining reasonable results when applying online anticipatory algorithms. If the input distribution of the arriving requests satisfies this requirement, it can also be characterised as *anticipative distribution*.

In the scope of this work the developed algorithms are also studied experimentally for the *packet scheduling problem*. The considered variation of the problem has been originally studied in [9] and further discussed in [1, 4, 10, 11]. The packet scheduling problem has many important practical applications, mainly for the management of real-time multimedia traffic, where the flow of packets across an IP network in communication networks is optimized [11]. It is a common situation, that large data frames are fragmented into smaller packets and sent individually through the network. If only a few of these packets are dropped, the remaining fragments of the entire data frame might be useless [12].

1.1 Online Algorithms

Traditional offline algorithms assume that the complete input is known. Based on this entire input, the output is generated. However, in practice this assumption often does not hold and one can observe only the revealed input so far. That is, the input is only partially observable at the time of a decision since information on future inputs is not revealed before their actual

arrival. These algorithmic problems are referred to as *online problems*.

Formulating this idea in a more mathematical way, online problems are usually described by an input sequence $I = (I(1), I(2), \ldots, I(h))$ that is presented to the online algorithm step by step over the horizon $T = (1, 2, \ldots, h)$. When dealing with the input $I(t)$, which is revealed at time t, no other input $I(t')$ is certainly known to the algorithm for all $t' > t$ [13].

1.1.1 Competitive Ratio

Online algorithms need to generate decisions based on incomplete information. The performance of such algorithms is in general assessed by using *competitive analysis*. For that an offline optimization algorithm $\mathcal{O}$ must be available, that given the entire input sequence from start produces an optimal decision sequence. Optimal in this context means, that its solution value $w(\mathcal{O}(I))$ must be larger or equal to any other valid solution $w(\mathcal{O}'(I))$, given the same input. Competitive analysis then compares the solution of the offline optimization algorithm on an input sequence I to the solution of the online algorithm $\mathcal{A}$ on the same input sequence. For the comparison the worst-case input sequence I is considered. Therefore, the performance can be measured by the *competitive ratio*, the maximum ratio (over all possible sequences I) between the value of the optimal solution $w(\mathcal{O}(I))$ and the solution value of the online algorithm $w(\mathcal{A}(I))$, that is

$$\max_I \frac{w(\mathcal{O}(I))}{w(\mathcal{A}(I))}. \tag{1.1}$$

Similar to the performance measurement of approximation algorithms [14], we can say algorithm $\mathcal{A}$ is c-competitive if

$$w(\mathcal{O}(I)) \leq cw(\mathcal{A}(I))$$

holds for all possible sequences I of the problem instance. Note that this definition is adequate for an online maximization problem, otherwise for an online minimization problem the algorithms $\mathcal{A}$ and $\mathcal{O}$ need to be exchanged in the definition above.

In practice, the bound provided by the competitive ratio is often very pessimistic since the competitive ratio is determined by the worst-case input sequence. It assumes that nothing at all is known about the distribution of the input sequence and therefore, that every possible sequence is to be expected. There are several alternative models, such as the *diffuse adversary model*, which samples the input sequences from several possible distributions [15]. An extension to the competitive ratio can be found in the context

of *randomized online algorithms*, a probability distribution over deterministic online algorithms. In this setting the solution on an input sequence must be described through its expected value and the algorithm $\mathcal{A}$ is c-competitive if

$$w(\mathcal{O}(I)) \leq c\mathbb{E}[w(\mathcal{A}(I))],$$

where in this definition the expectation regards the randomization of the algorithm, not the input sequences I [16].

1.1.2 Regret Minimization

The performance of online algorithms can also be analysed by *regret minimization*. Regret minimization again incorporates the comparison between the offline algorithm $\mathcal{O}$ and the online algorithm $\mathcal{A}$ by the definition of the regret to be the difference of the objective function w of the two decision sequences produced by either algorithm [17]. The effort here is of course to minimize regret and eventually to achieve a *vanishing average regret* which implies that the online algorithm attains the offline algorithm's performance [18]. This of course is not attainable in general, but e.g. when input sequences are drawn from a certain distribution $\mathcal{I}$. Given that the distribution $\mathcal{I}$ is anticipative for the problem, the expected regret is small. The definition of the anticipative distribution $\mathcal{I}$ can be found in Section 2.2.2. For the assessment of an online algorithm against the offline algorithm the average regret can be formulated as

$$L = \mathbb{E}_I[w(\mathcal{O}(I)) - w(\mathcal{A}(I))], \tag{1.2}$$

that is, the expected loss over all input sequences I drawn from the distribution $\mathcal{I}$ [1]. This approach is less pessimistic than the competitive ratio by the fact that it is not a worst-case scenario, but rather the consideration of an average loss.

1.2 Online Scheduling

This section defines the class of problems that are discussed in the scope of this work. In the following it is assumed that the problem to be solved is a *scheduling problem* within the online framework. Optimal scheduling is a significant field of operations research with wide-ranging practical applications [19], e.g. packet scheduling in network routers or machine scheduling in manufacturing.

The scheduling problem considered in this work is characterized by an input sequence that arrives over a finite horizon $T = (1, 2, \dots, h)$. The input

is a sequence $I = (R_1, \ldots, R_h)$ of request sets R_t, arriving at time $t \in T$. Each request set $R_t \in I$ can be empty or contain a finite number of requests $r \in R_t$. However, at each time step only one request can be served. The solution to the scheduling problem is a decision sequence $\boldsymbol{\gamma}$ that maximizes the problem specific objective function $w(\boldsymbol{\gamma})$ under some given constraints $H(\boldsymbol{\gamma})$. The decision sequence $\boldsymbol{\gamma}$ contains the request $\gamma_t \in I$ that is scheduled at time t.

Note that throughout this work sequences are treated like sets and operations from set theory are applied to these sequences. This has the advantage to be able to emphasize sequential behaviour, e.g. requests over time, while keeping the notation simple. The notation $R_t \in (R_1, \ldots, R_h)$ is therefore defined as $R_t \in \{R_1, \ldots, R_h\}$.

1.3 Anticipation

The definition and analysis of algorithms that incorporate anticipative behaviour in the previously discussed online framework is the fundamental objective of this work. The term *anticipation* is frequently used in the following and this section tries to give an intuitive understanding of it. A way of defining anticipative behaviour is that decisions do not only depend on the past and present, but also on predictions, expectations or beliefs about the future [6].

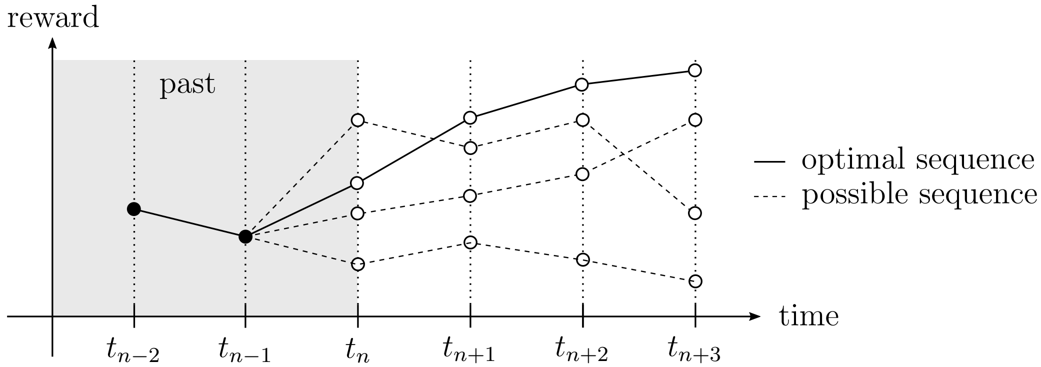


Figure 1.1: Optimal Decision Sequence with Anticipative Behaviour

Without any notion of the future, and therefore without any anticipative behaviour, decisions at a specific time t might be suboptimal, since future requests $R_{t'}$ with $t' > t$ are not observable. Therefore, this work considers algorithms that base their decisions on beliefs about the future using sampled possible requests. The reason why anticipative behaviour is useful in the online framework is illustrated in Figure 1.1. Intuitively, a decision that

might seem to be optimal at a specific time t, might be suboptimal when regarding the entire time horizon $t \in T$.

More precisely, a specific decision within a dynamic decision process contributes two aspects to the overall objective. The immediate contribution, meaning the immediate decision cost or reward, and the influence of the decision on the future. This influence can be caused due to constraints, e.g. deadlines, or a decision at a given time might exclude certain future decisions. Note that these two decision aspects are also included in the definition of the *return* (equation 3.1) in the reinforcement learning framework discussed in Chapter 3. There, the immediate reward is extended by discounted, expected future rewards. Anticipative algorithms aim to optimize the trade-off between these aspects in order to make optimal decisions over the entire horizon. This leads to the statement that optimal decisions can be taken, assuming perfect anticipation. This statement is discussed in detail throughout this work but note that perfect anticipation may be hard to achieve. It implies a tremendous number of iterations until convergence and significant memory requirements [3].

Chapter 2

Online Stochastic Optimization

Practical optimization problems often include stochastic uncertainties in the model parameters. In the procedure of decision-making they must be expressed using random variables having a certain probability distribution [20]. The uncertainty in online problems is mainly caused by the input sequences that are only revealed over time whereby their underlying distribution is either known a priori or must be learned over the time horizon.

Online stochastic optimization samples possible future requests from a distribution and uses either predictive models or historical data if necessary. If such models are available, the assumption of the availability of a distribution that can be sampled is reasonable in many contexts and the quality of solutions can be improved by using stochastic information [21].

This chapter discusses the stochastic approach to describe the uncertain inputs. The combination of stochastic programming and online algorithms introduces the concept of online anticipatory algorithms for decision-making under uncertainty [1].

2.1 Stochastic Programming

This section tries to give a brief introduction to stochastic programming, more precisely it introduces *multistage stochastic problems*. These are common in planning processes where decisions need to be made during subsequent instances of time, also called stages. These problems consist of a sequence of decisions and include uncertainty in their relevant parameters. In other words, a decision sequence needs to be taken without having full information on some random events [22]. This is also the case in the online framework, where at each time step a decision needs to be taken without knowledge of the future inputs.

In stochastic programming, the decisions under uncertainty are called *first-stage decisions*. Later, when full information on the actual realization of the random event is available, corrective or *second-stage* actions can be taken. For multistage stochastic problems, second-stage decisions can be taken at several times. Therefore, the decision process in a multistage stochastic program can be described by a sequence

$$x_0 \rightarrow \xi_1 \rightarrow x_1 \rightarrow \ldots \rightarrow x_{h-1} \rightarrow \xi_h \rightarrow x_h, \tag{2.1}$$

where x_t are decisions and ξ_t observations at a specific stage t. The multistage problem can either terminate with a decision (as shown in the sequence 2.1) or with an observation [23].

In this work *stochastic combinatorial optimization* is used to solve the multistage stochastic problem. This is further discussed in the subsequent sections. Note that stochastic programming is an a priori optimization method and it is therefore unlikely to be scaled to large horizons and applied in the online framework. But the most striking difference between stochastic combinatorial optimization and multistage stochastic programs is that the former does not incorporate second-stage actions. On the contrary, in the online framework decisions are usually irrevocable and cannot be corrected at later time steps [1].

2.2 Stochastic Combinatorial Optimization

As already stated in the first introductory paragraph of this chapter, the additional stochastic information can significantly improve the quality of solutions. Online anticipatory algorithms exploiting this information by sampling possible future requests can produce results close to the optimal a posteriori solution, assuming anticipativity. The *anticipativity assumption* implies that the order of the input sequence is not too significant and it can be shown experimentally that it holds for many applications [1]. This is a fundamental property in order to be able to apply online stochastic optimization to these problems.

Incorporating the stochastic information allows the online algorithm to solve the optimization problem in a similar way to the generic offline algorithm. A sequence of possible future requests is generated, which allows the application of existing offline optimization methods. This approach is called *stochastic combinatorial optimization*. However, in practical applications time might be limited for making a decision and additional constraints on the number of optimization steps at each time step might be given [10].

2.2.1 Sampling Future Tasks

Oblivious online algorithms try to optimize the problem specific cost function without incorporating any stochastic information or anticipative concepts. Implementations of such algorithms for the packet scheduling problem are discussed in Section 4.3. The anticipatory algorithms presented in this work extend this framework by incorporating anticipative behaviour by sampling the previously mentioned anticipative distribution $\mathcal{I}$ to generate a sequence of possible future inputs. This policy necessitates the implementation of a function SAMPLE that generates an input sequence

$$\textsc{sample}(t, h_s) = (\xi_{t+1}, \ldots, \xi_{t+h_s}),$$

based on the current time t and the sampling horizon h_s. The set of possible future inputs arriving at time t and sampled from the distribution $\mathcal{I}$ is denoted by ξ_t. This notation is used to distinguish uncertain future request sets $\xi_{t'}$ from certain request sets $R_{t''}$, that have already arrived at the current time t, where $t'' \leq t < t'$. Note that the sample horizon h_s needs to be specified since it is unrealistic to sample the future for the entire time horizon [1]. When optimizing a sampled scenario with the offline optimization algorithm, the known revealed requests R need to be concatenated with the sampled requests to produce a full input sequence

$$I_{\xi_{t+1}} = (R_1, \ldots, R_t, \xi_{t+1}, \ldots, \xi_{t+h_s}). \tag{2.2}$$

Here and in the following the context makes it necessary to use the notation $I_{\xi_{t+1}}$ to emphasize that the input sequence starting from time $t+1$ is based on sampled requests.

2.2.2 Anticipativity Assumption

The anticipativity assumption is the hypothesis that the distribution $\mathcal{I}$ is anticipative for the problem. This section also discusses how the anticipatory relaxation is used to solve the previously mentioned multistage stochastic program. An intuitive interpretation of the anticipativity assumption is that at each time step $t \in T$, there is a natural request r to select in order to maximize the expected objective [1]. This request must be a feasible request, meaning that it is a valid request to be processed at time t. Consistent with the definitions in Section 1.1, $\boldsymbol{\gamma}$ denotes the decision sequence and $H(\boldsymbol{\gamma})$ are the problem specific constraints. For better readability, here and in the following the notation $\Gamma_t \equiv (\gamma_1, \ldots, \gamma_t)$ is used to represent the sequence of past decisions made previously until the time step t. Before defining the set

of feasible requests, the definition of the set of all available requests at time step t is given by

$$\bigcup_{R_i \in I, i \leq t} R_i \setminus \Gamma_{t-1}.$$

Of course, the feasible requests must be a subset of all available requests and in addition to that, they also fulfil the problem specific constraints $H(\boldsymbol{\gamma})$ that define a valid solution. This allows us to define the set of feasible requests at time t as

$$\mathcal{F}(\Gamma_{t-1}, I) = \left\{ r \in \bigcup_{R_i \in I, i \leq t} R_i \setminus \Gamma_{t-1} \,\middle|\, H(\Gamma_{t-1} \cup \{r\}) \right\}, \tag{2.3}$$

where $\Gamma_{t-1} \cup \{r\}$ denotes the concatenation of the past decisions with the decision of selecting the request r at time t. Assuming the offline optimization algorithm $\mathcal{O}'$ is given, that solves problem $\mathcal{P}$ given a sequence of past decisions, a distribution $\mathcal{I}$ is anticipative if

$$\mathop{\mathbb{E}}_{I_{\xi_t}} \left[w(\mathcal{O}'(\Gamma_{t-1}, I_{\xi_t})) \right] = \mathop{\mathbb{E}}_{I_{\xi_t}} \left[\max_{\gamma_t \in \mathcal{F}(\Gamma_{t-1}, I_{\xi_t})} \mathop{\mathbb{E}}_{I_{\xi_{t+1}}} \left[w(\mathcal{O}'(\Gamma_t, I_{\xi_{t+1}})) \right] \right] \tag{2.4}$$

holds for all $t \in T$ and realizations R_t of ξ_t [1]. The difference of the offline optimization algorithm $\mathcal{O}'$, further defined in Section 4.3.2, to the algorithm $\mathcal{O}$ used previously, is that $\mathcal{O}'$ considers already scheduled tasks as a constraint. On the left side of this equation the offline optimization algorithm is applied to the input sequence I_{ξ_t}, defined in equation 2.2. On the right side of equation 2.4 the input sequence I_{ξ_t} is also drawn from the distribution $\mathcal{I}$, but it is used to calculate the set of feasible decisions. For each such decision γ_t, the next input sequence $I_{\xi_{t+1}}$ is then drawn from $\mathcal{I}$, assuming that the set of requests in time t is given by $R_t = \xi_t$. This information is then used to optimize the scenario with the offline optimization algorithm $\mathcal{O}'$.

The anticipativity assumption therefore states, that the expected value of the solution of $\mathcal{O}'$ over all I_{ξ_t} drawn from distribution $\mathcal{I}$ is equivalent to the expected value, over all I_{ξ_t} drawn from $\mathcal{I}$, of the maximum expected value (over all feasible decisions γ_t based on the sequence I_{ξ_t} and past decisions Γ_{t-1}) of the solution of $\mathcal{O}'$ over all $I_{\xi_{t+1}}$ drawn from $\mathcal{I}$ under the assumption that $R_t = \xi_t$. By induction on t this assumption can be extended to the time horizon h requiring that, at each time step t, the *anticipatory relaxation*

$$\max_{\gamma_t \in \mathcal{F}(\Gamma_{t-1}, I)} \mathop{\mathbb{E}}_{I_{\xi_{t+1}}} \left[\max_{\gamma_{t+1} \in \mathcal{F}(\Gamma_t, I_{\xi_{t+1}})} \cdots \max_{\gamma_h \in \mathcal{F}(\Gamma_{h-1}, I_{\xi_{t+1}})} w(\Gamma_h) \right] \tag{2.5}$$

is equivalent to the multistage stochastic program. This multistage problem is typically very challenging and contains a large number of stages [1].

2.3 Online Anticipatory Algorithms

The introduction of online algorithms and the stochastic approach to generate possible future requests allows the combination of these concepts to eventually define *online anticipatory algorithms*. It combines all the previous aspects discussed in this chapter and makes decisions online based on samples of the anticipative distribution $\mathcal{I}$. The process of making a decision online in these algorithms can be divided into three main steps [4]:

1. sample the available anticipative distribution $\mathcal{I}$ to generate possible future scenarios;

2. compute the optimal decision for each scenario;

3. select a decision.

It can be noticed that the steps of the online anticipatory algorithms are based on the anticipativity assumption. The steps shown above are derived from the mathematical formulation of the right-hand side of the equation for the anticipativity assumption defined in equation 2.4.

Note that step 1 corresponds to the idea discussed in Section 2.2.1 and uses the function SAMPLE to obtain possible future scenarios. Step 2 uses the offline optimization algorithm $\mathcal{O}'$, constrained by the past decisions, which is presented in Section 4.3.2 for the packet scheduling problem. In Section 4.4, implementations of online anticipatory algorithms are presented. This work discusses three different approaches to such algorithms in the context of the packet scheduling problem. The first one is the expectation approach $\mathcal{E}$, the basic online anticipatory algorithm that considers each possible decision at each step. As mentioned earlier, time might be constricted and only few optimization steps might be feasible. Therefore, two approximations of $\mathcal{E}$, the consensus approach $\mathcal{C}$ and the regret approach $\mathcal{R}$, are also discussed in Section 4.4.

A related concept to the approach presented in this chapter is *evolutionary online optimization* [24, 25]. It considers the same idea of sampling the underlying distribution $\mathcal{I}$, but instead of solving the optimization problem at each time step by an offline optimization algorithm, a population of algorithms is maintained and used for optimization. The best result of any algorithm is used to choose the request at time t and afterwards genetic operators are applied on the population to add new algorithms to the population. This concept may be promising for problems that are NP-hard, where executing an offline optimization algorithm might be infeasible [25]. However, this approach is not further discussed in the scope of this work.

Chapter 3

Reinforcement Learning

In the previous chapter, an algorithmic framework to solve problems that involve sequential decision-making in the online framework has been presented. This chapter considers such problems by using a fundamentally different approach. A common way to describe sequential decision-making settings can be found in the framework of *Markov decision processes* [1, 26, 27]. A standard approach to solve such problems offline is dynamic programming, where the model must be known in advance.

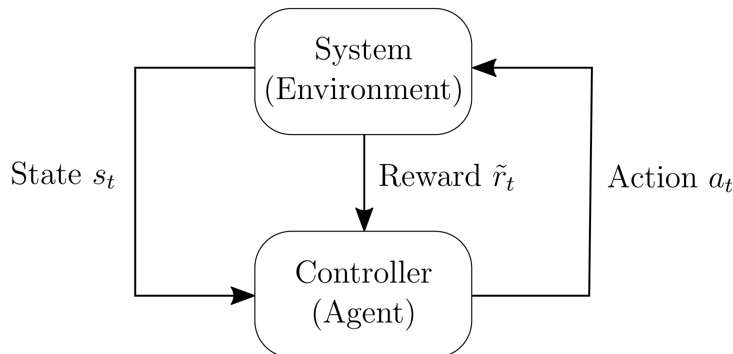


Figure 3.1: Reinforcement Learning Scenario

This section discusses how to solve the Markov decision process in an online setting by using *reinforcement learning* [26]. Contrary to dynamic programming, reinforcement learning does not require the knowledge of the model in advance. It has already been shown that reinforcement learning can be successfully applied to a wide range of real-world online stochastic problems, e.g. vehicle routing, bin packing [28] and also several variants of the packet scheduling problem [29].

Reinforcement learning is a learning setting where the goal is to maximise some numerical value that represents the overall objective. The idea is to

learn through interaction with an environment, modelled by a Markov decision process, in a goal-oriented manner. Intuitively, reinforcement learning can be described by learning what to do (mapping situations to actions) by discovering what actions are most profitable in a given situation by actually trying them when interacting with the system [30]. A typical reinforcement learning scenario is depicted in Figure 3.1. An agent observes the state of the environment together with a reward determined by the last state transition. Based on this information, the agent must choose the current action which is then sent to back to the system. Afterwards the cycle is repeated. In this way, reinforcement learning algorithms by nature are online algorithms for solving Markov decision processes [31] and can therefore be applied in the online framework without the necessity of adaptations.

3.1 Finite Markov Decision Processes

This first section now discusses the Markov decision process in more detail. As already mentioned, Markov decision processes describe sequential decision-making settings. What makes this framework interesting in the context of this work is that actions not only influence the immediate reward, but also future situations and therefore future rewards. Thus, Markov decision processes involve delayed reward [30]. The concepts discussed in this section still regard the offline setting where the optimal policy π^* is defined based on the known model, including rewards and transition probabilities. A policy is a strategy that defines how the agent behaves at a given time by mapping states to actions.

Each Markov decision process consists of states $s \in S$, actions $a \in A$, the transition function p and a reward function $\tilde{r}$. This work only considers *finite Markov decision processes*, where the sets of states and actions are all finite sets (and therefore contain only a finite number of elements). At each time step t, the agent interacts with its environment and chooses an action a in the observed state s. The transition function $p(s, a, s')$ then describes the probability of arriving in state s' after choosing action a in the previous state s. For Markov decision processes the environment is *Markovian* and therefore the result of an action only depends on the current state (and not on previous actions or the state history), more precisely

$$P(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \ldots) = P(s_{t+1}|s_t, a_t) = p(s_t, a_t, s_{t+1}).$$

This can also be viewed as a restriction on the states such that each state must contain all information on the environment that could influence a possible future state [27].

3.1.1 Reward Signal and Return

The idea to formulate the objective by using a reward signal is a key feature of the online reinforcement learning task [30]. The reward signal is transmitted from the environment to the agent after each action it has taken. It is a simple number, but it is crucial that the reward signal precisely indicates what is desired to be accomplished. The reward signal must not contain any details on how to achieve a goal (e.g. rewards for achieving subgoals), otherwise it might happen that the agent learns to cumulate reward without even actually reaching the real goal. It is also important to note that in any case the agent must aim to maximise the cumulative reward and not the immediate reward. This idea is again fundamental to be able to incorporate any anticipative behaviour as explained in Section 1.3. The reward signal therefore evaluates an action taken in a specific state. However, despite the term *reward* that might indicate a positive signal, the reward signal is scalar. It can be also negative which indicates a punishment or cost for a certain action.

In general, when selecting an action a at time step t, the action shall be chosen such that the *expected return* is maximised. In a simple case the return could be defined as the sum of the rewards, however, when there is no terminal state the return could be infinite. To overcome this limitation the concept of *discounting* can be used. By that, the action is chosen based on maximising the expected discounted return

$$G_t = \tilde{r}_{t+1} + \rho\, \tilde{r}_{t+2} + \rho^2 \tilde{r}_{t+3} + \ldots = \sum_{k=0}^{\infty} \rho^k \tilde{r}_{t+k+1}, \tag{3.1}$$

where ρ is the discount rate. If $\rho = 0$ the agent only maximises the immediate reward and as ρ gets closer to 1, future rewards are taken more and more into account [30].

3.1.2 Optimal Value Function

Given a Markov decision process, an optimal policy π^* that accumulates maximum discounted return can be defined via *value functions*. The value function $v_\pi(s)$ calculates the expected return when starting in state s and following policy π afterwards. It can be written as

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid s_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \rho^k \tilde{r}_{t+k+1} \,\middle|\, s_t = s \right] \tag{3.2}$$

for all $s \in S$. We can thus say, policy π is better than policy π', if

$$\forall s \in S : v_\pi(s) \geq v_{\pi'}(s).$$

The optimal policy π^* must therefore be better than all other policies, although there might be more than one optimal policy. All optimal policies must have the same optimal value function v_*, defined as

$$v_*(s) = \max_\pi v_\pi(s).$$

In addition, the *action-value function* $q_\pi(s, a)$ for policy π is the value of taking action a in state s and following policy π afterwards. It is defined as

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid s_t = s, a_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \rho^k \tilde{r}_{t+k+1} \middle| s_t = s, a_t = a \right],$$

that is, the expected return starting from state s with action a being taken and then following policy π. Again, all optimal policies share the same optimal action-value function q_*, defined as

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

for all $s \in S$ and all $a \in A$. Note that the optimal action-value function for a state-action pair (s, a) can be written as

$$q_*(s, a) = \mathbb{E} \left[\tilde{r}_{t+1} + \rho v_*(s_{t+1}) \mid s_t = s, a_t = a \right], \tag{3.3}$$

that is, the expected return for taking a specific action a in the state s and afterwards following the optimal policy.

3.2 Q-Learning

This section now transfers the previously discussed framework of Markov decision processes into the online framework. The *Q-learning* [32] algorithm is presented, that can be used to solve a reinforcement learning task. Q-learning is a basic and popular method, part of the larger class of model-free *temporal difference* learning algorithms. That means, that contrary to dynamic programming, these algorithms do not build a model of the Markov decision process and are able to learn directly from the raw experience without having any notion of the system they are applied on. Furthermore, a key feature of temporal difference learning algorithms is the gradual update

of estimations, that themselves are based on other estimations (also called *bootstrapping*) [27].

The basic idea of Q-learning is to gradually estimate the action-value function Q and by that, directly approximating the optimal action value function q_*. The update rule of the Q-learning algorithm is given by

$$Q(s_t, a_t) = Q(s_t, a_t) + \lambda \left(\tilde{r}_{t+1} + \rho \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right), \tag{3.4}$$

where λ is a learning parameter. This simple update rule also significantly simplifies the analysis of the algorithm. It has been shown that with Q-learning the learned action-value function Q converges with probability 1 to the optimal action-value function q_*, under the assumption that all state-action pairs continue to be visited (and therefore updated) [33]. In other words, Q-learning converges when each state-action pair is visited infinitely often.

3.3 Exploration and Exploitation

Q-learning allows us to approximate the optimal action-value function q_* and it is guaranteed to converge, but only on the condition that each state-action pair is visited infinitely often. However, it can happen that some states stop being visited due to the current estimation of the Q-function, therefore the algorithm can be stuck in a local optimum. Choosing actions only based on Q (*exploitation*) is thus not always expedient. To handle this problem, *exploration* is necessary. However, exploring too much will accumulate a large loss over time and balancing exploration and exploitation is a key problem in reinforcement learning, also known as the exploration-exploitation dilemma [34].

The obvious conclusion is that exploration (choosing untested actions) and exploitation (choosing actions that are known to be good) must happen simultaneously. There are two types of exploration that can be distinguished: *directed* and *undirected* exploration. Undirected exploration is a simple exploration based on randomness, e.g., ϵ-*greedy* [32] or *Boltzmann exploration* [30]. Directed exploration on the other hand keeps track of some additional information, e.g. how many times each state-action pair has been visited. Bayesian Q-learning [35] or the UCB-1 algorithm [36] are examples for incorporating directed exploration.

3.3.1 Epsilon Greedy

The directed exploration strategy ϵ-greedy is one of the most used methods [34]. It is also referred to as *semi-uniform random distribution* [35], where with a specific probability ϵ a suboptimal (according to the current action-value function Q) action is chosen. The parameter $0 \leq \epsilon \leq 1$ therefore indicates how much exploration takes place and how often (with probability $1-\epsilon$) the action with highest Q-value is chosen. It is common to gradually decrease the parameter ϵ with increasing time to reduce the loss accumulated by the exploration, once the function Q is expected to be a good approximation of the optimal action-value function q_*.

3.3.2 UCB-1

The UCB-1 method was originally developed for the multi-armed bandit problem [36] but can be adapted to be used for the general reinforcement learning problem. For that, the algorithm UCRL2 has been defined to use upper confidence bounds to choose an optimistic policy [37]. In this work this approach is not discussed, rather simple steps are taken to use upper confidence intervals directly with Q-learning [34]. The basic idea is to estimate an upper confidence $U(a)$ for each action value, such that with high probability

$$q_*(s_t, a_t) \leq Q(s_t, a_t) + U(a_t).$$

This can be done by keeping count of the number $N(s, a)$ of times each action $a \in A$ has been chosen in state s. The action is then chosen such that the upper confidence bound is maximised, that is,

$$a_t = \arg\max_{a \in A} \left(Q(s_t, a) + \sqrt{\frac{c \log N(s_t)}{N(s_t, a)}} \right), \tag{3.5}$$

where c is a parameter to control exploration and $N(s)$ is the number of times that state s has been visited. It is given by

$$N(s) = \sum_{a \in A} N(s, a).$$

Setting the parameter c larger leads to an increase in exploration. Also, a small $N(s, a)$ leads to a larger upper confidence value $U(a)$, that means the estimated value is uncertain [36, 34].

3.4 Non-Markovian Observations

Previously, in Section 3.1, Markov decision processes have been defined. A key feature of them is their Markovian environment, where each state must contain all information necessary to describe the environment. For the online scheduling problems discussed in this work, the arriving input sequences can be included in the state representation, however it cannot be assumed that the Markov property is still satisfied. By that, the arriving input sequences may be *non-Markovian* observations. Imagine the input distribution $\mathcal{I}$ to be a simple Markov model with more than one state (different from the states of the Markov decision process). Since the states of the Markov model cannot be observed, they also cannot be included in the state representation of the Markov decision process.

It is possible to simply ignore the violation of the Markov property, but it is not guaranteed that the estimated Q-function will then lead to a reasonable policy. Consider again the example above. If the fact that the input distribution has several states is ignored, the action-value function of the state-action pair is calculated over all these states of the input distribution, which might not be expedient. Another approach is to change the problem such that the environment does not emit its states, but only the actual observations. That is, the agent receives signals that depend on the current state, but only provide partial information about it. In this case a simple option is to assume that the reward function is a direct function of the observations, since the actual states are not observable [30]. A more sophisticated approach would be to work with *partially observable Markov decision processes* that were introduced for that setting [38].

3.4.1 Conservative Q-Learning

The first approach to cope with the non-Markovian environment is the definition of a *conservative Q-learning* strategy. The problem with the basic update rule of the Q-learning algorithm (equation 3.4) in a non-Markovian environment is that the state s_{t+1} cannot be assumed to be inferred correctly. Therefore, the term

$$\max_a Q(s_{t+1}, a) - Q(s_t, a_t) \tag{3.6}$$

can also not be computed. A conservative Q-learning approach can be found by establishing a lower bound on the term shown in equation 3.6, that is, the right part of the Q-learning update rule for any state that could be reached (equation 3.4). By that, an overestimation of the Q-value can be

prevented [39]. If we consider the scheduling problem, a state changes with the arrival of a new request set. All other previously arrived requests however remain also open to be scheduled at time $t + 1$, given they still meet the constraint. These requests can be used to define the lower bound on the Q-update, since it is certain that these requests will be available to be scheduled in the next state. This allows to define the update rule of the conservative Q-learning by

$$Q(s_t, a_t) = Q(s_t, a_t) + \lambda \left(\tilde{r}_{t+1} + \rho \max_{r \in \mathcal{F}_{t+1}} w(r) \right), \tag{3.7}$$

where $\mathcal{F}_{t+1}$ is the set of feasible requests at time $t+1$, defined in equation 2.3. By this update rule, the algorithm only uses information that is completely certain. Note, that the reward $\tilde{r}_{t+1}$ is also independent of the arriving requests at time $t + 1$. It only depends on the objective function of the request scheduled at time t, reduced by the objective function of requests that are lost in the next time step due to the constraints. It is important that the loss is now explicitly modelled in the reward function, since no long-term consequences are considered by the conservative Q-learning approach.

3.4.2 Belief States

The second approach presented in this work to deal with non-Markovian observations is to use the environment's history, that is, the input sequence I observed until the current time t. Note that the state representation is a function of the history, that is, $s_t = f(I)$. By that, it is possible to use the past to predict the missing information of the environment, that is, the current state of the underlying distribution. For the case that the input distribution $\mathcal{I}$ is described by a Markov model, the past can be used to infer the current Markov state (note the difference between states in the reinforcement learning algorithm and Markov states), which then can be included in the state representation of the reinforcement learning algorithm. The estimated Markov states of a Markov model are also called *belief states*. Doing so will allow the system to meet the Markov property, assuming the belief states are inferred correctly. A technique to estimate the current belief state can be found in Section 5.1.

Chapter 4

Online Packet Scheduling

In this chapter the theoretical concept of online stochastic optimization from the previous Chapter 2 is studied under the aspect of the *online packet scheduling* problem. A convenient property of this problem is that its offline version can be solved in polynomial time given fixed deadlines d and fixed processing time. The more general packet scheduling problem with variable deadlines and processing times is known to be NP-hard [40]. In practice, online packet scheduling has an important application in communication networks for optimizing the flow of packets across an IP network [11].

A detailed definition of the problem is given in the first section of this chapter. The subsequent sections discuss algorithms for offline optimization together with algorithms that can be applied in the online framework and implement the concept of online anticipatory algorithms.

4.1 Problem Definition

Online packet scheduling considers the scheduling of a sequence of packets. Several assumptions are made to simplify and further specify the problem. It is assumed that each packet corresponds to a specific class c from a finite set of classes $c \in C$. This class determines the packet's reward $w_c > 0$, also referred to as its weight. Further it is assumed that each packet takes the same processing time of one time step and time t is discrete over the time horizon $t \in T = \{1, \ldots, h\}$. It is also not allowed for multiple packets of the same class to arrive at the same time. Therefore, each packet can uniquely be identified by its class and arrival time. At last it is assumed that each packet has the same deadline d relative to its arrival time [9]. The packets j arrive as an input sequence of sets of packets $I = (R_1, \ldots, R_h)$, where each packet $j \in R_t$ has arrival time $a(j) = t$. Each packet j must be scheduled

within its time window $[a(j), a(j) + d]$ and not more than one packet can be scheduled at each time t. In a more mathematical formulation, the objective function $w(\boldsymbol{\gamma})$ that needs to be maximized is the sum of the weights of all scheduled packets, that is

$$w(\boldsymbol{\gamma}) = \sum_{t \in T} w(\gamma_t), \tag{4.1}$$

where γ_t is the packet j which is scheduled at time t. The maximization is constrained by the constraints $H(\boldsymbol{\gamma})$ on $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_h)$, specified as

$$H(\boldsymbol{\gamma}) \equiv \forall t \in T : a(\gamma_t) \leq t \leq a(\gamma_t) + d, \tag{4.2}$$

that forces each packet to be scheduled within its time window [1].

4.2 Offline Optimization

Stochastic sampling generates possible future requests and allows us to solve the online decision problem by optimization in an offline fashion. The offline optimization algorithm $\mathcal{O}$ is shown in Figure 4.1 and implements step 2 of online anticipatory algorithms presented in Section 2.3. However, the aspect that in an offline setting several possible solutions might be optimal, must receive attention. In an offline solution packages can be exchanged without any cost changes but in an online setting these solutions are not equivalent. High-weight packets should be scheduled early in order to avoid the risk of losing these packets. Since in the offline framework these solutions are equivalent, also their objective value is equivalent. Therefore, a postprocessing step can be added to the offline algorithm to implement the early scheduling of high-weight packets [1].

Algorithm $\mathcal{O}$ schedules the tasks starting with packets from the class c of highest weight w_c to the classes with lowest weight, where the packets j with latest arrival $a(j)$ are prioritized. If a packet cannot be added to the schedule because there is no free slot during its time window, other packets are shuffled to try to make room for the new packet if possible. The goal of this process is generating a minimum loss schedule, that is scheduling the packets such that the cumulative weight of all lost packets is minimized [9].

Therefore, $\mathcal{O}$ schedules each packet j greedily by the policy described in the previous paragraph to a time step t^* as late as possible before its deadline $a(j)+d$ (lines 7 and 9). If no time $t \in T$ exists to schedule packet j, it is ignored (line 8). Otherwise, if scheduling j at time t^* satisfies the constraints H, specified by equation 4.2, packet j is scheduled at t^* (line 7).

In the case that $t^* < a(j)$ algorithm $\mathcal{O}$ tries to find the earliest packet k scheduled at time t' with $w(k) \geq w(j)$ that can be scheduled at time t^* (lines 21 and 23). This is done by calling the function SHUFFLE in line 13.

Algorithm $\mathcal{O}(I)$

```
1: R ← ∪_{R_t ∈ I} R_t
2: for t ∈ T do
3:     γ_t ← null
4: end for
5: order R decreasing by w(j) and a(j)
6: for j ∈ R do
7:     S ← {t ∈ T | t ≤ a(j) + d and γ_t = null}
8:     if S ≠ {} then
9:         t* ← max(S)
10:        if t* ≥ a(j) then
11:            γ_{t*} ← j
12:        else
13:            γ ← SHUFFLE(γ, j, t*)
14:        end if
15:    end if
16: end for
17: return γ

SHUFFLE(γ, j, t*)
18: γ' ← γ
19: γ_{t*} ← j
20: while t* < a(j) do
21:     S ← {t' ∈ T | t* + 1 ≤ t' ≤ t* + d and a(γ_{t'}) ≤ t*}
22:     if S ≠ {} then
23:         t** ← min(S)
24:         swap packets γ_{t*} and γ_{t**} in γ
25:         t* ← t**
26:     else
27:         return γ'
28:     end if
29: end while
30: return γ
```

Figure 4.1: Offline Optimization Algorithm $\mathcal{O}$

The requirement $w(k) \geq w(j)$ is already fulfilled due to the greedy schedul-

ing, so that each packet k that has been scheduled previously to packet j must have at least weight $w(j)$. If such a packet k originally scheduled at time t^{**} exists, it is scheduled at slot t^* and packet j at slot t^{**} (line 24). The shuffling process is then repeated until either packet j is validly scheduled after its arrival date $a(j)$ or no further packet can be swapped. In the second case the original, unaltered schedule is returned.

Algorithm $\mathcal{O}$ has a runtime with complexity $O(h\,|C|)$, where h is the time horizon and $|C|$ the number of distinct classes [1].

4.2.1 Postprocessing

As mentioned in the beginning of this section, it is a good idea to add a postprocessing step to the offline algorithm to force scheduling high-weight packets early. Later in Chapter 6 it is shown that this can improve the solutions of the local optimization $\mathcal{L}$, presented in Figure 4.4 below, and the solution of all online anticipatory algorithms defined in Section 4.4. The idea is to take the optimal solution $\boldsymbol{\gamma}$ of the offline algorithm $\mathcal{O}$ and rearrange it to produce an equivalent solution in the offline context, that is advantageous in the online context.

Algorithm POSTPROCESS($\boldsymbol{\gamma}$)

1: **for** $t \in T$ **do**
2:     **for** $t' \in T \mid t < t'$ **do**
3:         **if** $(a(\gamma_t) \leq t' \leq a(\gamma_t) + d) \wedge (a(\gamma_{t'}) \leq t \leq a(\gamma_{t'}) + d)$ **then**
4:             **if** $w(\gamma_t) < w(\gamma_{t'})$ **then**
5:                 swap packets γ_t and $\gamma_{t'}$ in $\boldsymbol{\gamma}$
6:             **else if** $w(\gamma_t) = w(\gamma_{t'})$ and $a(\gamma_t) > a(\gamma_{t'})$ **then**
7:                 swap packets γ_t and $\gamma_{t'}$ in $\boldsymbol{\gamma}$
8:             **end if**
9:         **end if**
10:     **end for**
11: **end for**
12: **return** $\boldsymbol{\gamma}$

Figure 4.2: Postprocessing Step of the Offline Optimal Solution

The postprocessing algorithm shown above in Figure 4.2 iterates over each pair $(\gamma_t, \gamma_{t'})$ with $t < t'$ and, if possible, swaps the two packets if γ_t has a smaller weight (line 5) or the same weight and a later deadline (line 7). A swap is possible if the resulting configuration doesn't violate the con-

straints $H(\boldsymbol{\gamma})$ (line 3), therefore if

$$(a(\gamma_t) \leq t' \leq a(\gamma_t) + d) \wedge (a(\gamma_{t'}) \leq t \leq a(\gamma_{t'}) + d).$$

Since no packets are swapped if H is violated afterwards, no packet is lost and that means the solution is equivalent in the offline context. However, the new solution incorporates our effort to schedule high-weight packets early. It can therefore be said that the postprocessed solution is better in the online context since fewer high-weight packets are expected to be lost due to uncertainty in the inputs [1].

4.3 Oblivious Online Packet Scheduling

Section 2.3 introduced the online anticipatory algorithms studied in this work incorporating stochastic sampling in an online framework. These algorithms fall into the category of *adaptive algorithms*, whereby *oblivious algorithms* are the contrary approach. Oblivious algorithms do not include any anticipative behaviour. They are attractive since they are easy to implement and usually faster, however, their result is expected to be worse compared to adaptive algorithms [41].

This section further pursues this topic by introducing two oblivious implementations of online algorithms, the greedy algorithm $\mathcal{G}$ and the local optimization algorithm $\mathcal{L}$ in the context of the packet scheduling problem. These two algorithms implement the structure of the *generic online algorithm* which is presented in the next section.

4.3.1 Greedy Algorithm

The basic concept of the greedy algorithm $\mathcal{G}$ is very simple and its implementation is presented below in Figure 4.3. A greedy algorithm always makes the choice that looks best at the moment, which might however be a suboptimal decision globally [42].

For the packet scheduling problem, the greedy algorithm always chooses at each time step t the packet j with the highest weight from the set of available requests R to be scheduled at γ_t (line 8). However, the chosen request must be a valid request and therefore be in the set of feasible requests $j \in \mathcal{F}$ defined by equation 2.3 (line 7). Therefore, the information $(\gamma_1, \ldots, \gamma_{t-1})$ is used to avoid scheduling the same packet twice in different time steps which would result in an invalid solution. In this way, past decisions restrict the scheduling at the current time t. The past decisions are therefore taken into

consideration, but there is any anticipative behaviour, rather this information is only used to produce feasible solutions.

Algorithm $\mathcal{G}(I)$

1: **for** $t \in T$ **do**
2: $\quad \gamma_t \leftarrow null$
3: **end for**
4: **for** $t \in T$ **do**
5: $\quad R \leftarrow \cup_{R_{t'} \in I, t' \leq t} R_{t'}$
6: $\quad \Gamma_{t-1} \leftarrow \{\gamma_1, \ldots, \gamma_{t-1}\}$
7: $\quad \mathcal{F} \leftarrow \{j \in R \setminus \boldsymbol{\gamma} \mid H(\Gamma_{t-1} \cup \{j\})\}$
8: $\quad \gamma_t \leftarrow \arg\max_{j \in \mathcal{F}}(w(j))$
9: **end for**
10: **return** $\boldsymbol{\gamma}$

Figure 4.3: Online Greedy Algorithm $\mathcal{G}$

This implementation again illustrates that the oblivious algorithm does not attempt to predict information on the future.

4.3.2 Local Optimization

Another oblivious online algorithm is the local optimization algorithm $\mathcal{L}$ shown in Figure 4.4. Like the greedy algorithm it does not incorporate any anticipative behaviour.

Algorithm $\mathcal{L}(I)$

1: **for** $t \in T$ **do**
2: $\quad \gamma_t \leftarrow null$
3: **end for**
4: **for** $t \in T$ **do**
5: $\quad R \leftarrow \cup_{R_{t'} \in I, t' \leq t} R_{t'}$
6: $\quad \boldsymbol{\gamma}^* \leftarrow \mathcal{O}'(\boldsymbol{\gamma}, R)$
7: $\quad \gamma_t \leftarrow \gamma_t^*$
8: **end for**
9: **return** $\boldsymbol{\gamma}$

Figure 4.4: Online Local Optimization Algorithm $\mathcal{L}$

However, algorithm $\mathcal{L}$ does not simply choose the packet with highest weight at each time step t as the greedy algorithm does, rather it performs an

optimization on the known requests $(R_1, \ldots, R_t)$ for each time step (line 6). This optimization is a local optimization in the sense that it is applied on an incomplete input sequence only consisting of the requests up to the current time step t. In this way it chooses a locally optimal choice in each step hoping that this will also result in a globally optimal decision strategy (line 7). Note that the offline optimization algorithm $\mathcal{O}'$ is slightly different to algorithm $\mathcal{O}$ shown in Figure 4.1. It is again necessary to incorporate the information on the past to produce valid solutions without scheduling the same packet twice. Therefore, algorithm $\mathcal{O}'$, besides the request sequence I (or R in figure 4.4), also takes the current schedule $\boldsymbol{\gamma}$ as argument that contains all scheduled packets up to this time step t, meaning that $\gamma_{t'} = null$ for all $t' \geq t$. Algorithm $\mathcal{O}'$ results from algorithm $\mathcal{O}$, making the following two changes:

1. remove the for-loop that initializes $\boldsymbol{\gamma}$ starting at line 2 in Figure 4.1;
2. don't consider the packages $j \in \boldsymbol{\gamma}$ already included in the latest schedule when ordering the packages at line 5 in Figure 4.1.

4.4 Anticipative Online Packet Scheduling

In this section, several implementations of online anticipatory algorithms are presented. They incorporate anticipative behaviour by learning information from the experienced past when observing the input sequence I. An essential part of this work is the analysis and assessment of these algorithms applied to the packet scheduling problem. All online algorithms presented here have a basic structure that is shown in Figure 4.5 below.

Algorithm $\mathcal{A}(I)$
1: **for** $t \in T$ **do**
2: $\quad \gamma_t \leftarrow null$
3: **end for**
4: **for** $t \in T$ **do**
5: $\quad R \leftarrow \cup_{R_{t'} \in I, t' \leq t} R_{t'}$
6: $\quad \gamma_t \leftarrow \text{SELECT}(\boldsymbol{\gamma}, R, t)$
7: **end for**
8: **return** $\boldsymbol{\gamma}$

Figure 4.5: Generic Online Algorithm $\mathcal{A}$

The algorithm iterates over time steps t (line 1) and uses the known requests up to t (line 5) together with the past decisions to select the new packet to

be scheduled at t (line 6). When looking into the implementations of the greedy algorithm $\mathcal{G}$ and the local optimization algorithm $\mathcal{L}$, it can be noticed that also these oblivious online algorithms inherit this generic structure. The function SELECT (line 6) specifies the algorithm and for online anticipatory algorithms it implements the basic steps defined in Section 2.3. That means it uses the function SAMPLE and the offline optimization algorithm $\mathcal{O}'$ to select the next packet to be scheduled at time t.

4.4.1 Expectation Algorithm

The first algorithm studied in this section is the *expectation algorithm* $\mathcal{E}$. Its implementation is shown below in Figure 4.6. Obviously it samples future requests from the distribution $\mathcal{I}$ to generate possible future scenarios, but its distinctive property is that it evaluates each feasible request against each scenario. This is only reasonable when time is not considerably limited and numerous optimization steps can be performed.

Algorithm $\mathcal{E}(I, n)$

1: identical to algorithm $\mathcal{A}$ (see Figure 4.5) but SELECT-$\mathcal{E}$ is used in line 6

SELECT-$\mathcal{E}(\boldsymbol{\gamma}, R, t, n)$

2: $\Gamma_{t-1} \leftarrow \{\gamma_1, \ldots, \gamma_{t-1}\}$
3: $\mathcal{F} \leftarrow \{j \in R \setminus \boldsymbol{\gamma} \mid H(\Gamma_{t-1} \cup \{j\})\}$
4: **for** $j \in \mathcal{F}$ **do**
5: $\quad f(j) \leftarrow 0$
6: **end for**
7: **for** $i \leftarrow 1 \ldots \lfloor n/|\mathcal{F}| \rfloor$ **do**
8: $\quad S_F \leftarrow \{R \cup \text{SAMPLE}(t, h_s)\}$
9: $\quad$ **for** $j \in \mathcal{F}$ **do**
10: $\quad\quad \gamma_t \leftarrow j$
11: $\quad\quad f(j) \leftarrow f(j) + w(\mathcal{O}'(\boldsymbol{\gamma}, S_F))$
12: $\quad\quad \gamma_t \leftarrow null$
13: $\quad$ **end for**
14: **end for**
15: **return** $\arg\max_{j \in \mathcal{F}}(f(j))$

Figure 4.6: Expectation Algorithm $\mathcal{E}$

The main iteration of $\mathcal{E}$ is identical to the generic online algorithm discussed previously, so here and in the next sections only the function SELECT is discussed in detail for the respective algorithm. The basic idea of algorithm $\mathcal{E}$

is to choose the packet which leads to the solution with highest weight over the sampled scenarios. The first step to select the packet to be scheduled at time t is to calculate the set of feasible requests $\mathcal{F}$ (line 3). The parameter n is the total number of times the optimization algorithm $\mathcal{O}'$ is performed in line 11. Therefore, since each request $j \in \mathcal{F}$ is evaluated against each scenario S_F, exactly $k = \lfloor n/|\mathcal{F}| \rfloor$ scenarios are sampled (line 8), where $|\mathcal{F}|$ is the number of feasible packets. Of course, $|\mathcal{F}|$ will be different for each time step t and could also potentially be zero. This case needs to be treated separately to avoid an undefined division. Each scenario is then evaluated for each feasible request and the weight of the solution is added to the evaluation function f (line 11). Note, that $w(\mathcal{O}')$ is the objective function $w(\boldsymbol{\gamma})$, since the algorithm $\mathcal{O}'$ returns an optimal schedule based on the sampled scenario and considering the packets that already have been scheduled in the past. The decision of the algorithm $\mathcal{E}$ is based on the request with the highest evaluation function $f(j)$ (line 15).

At time t the decision γ_t can be computed using the anticipatory relaxation defined in equation 2.5. $\mathcal{E}$ approximates this equation by sampling the distribution $\mathcal{I}$ exactly k times to solve the deterministic problem

$$\max_{\gamma_t \in \mathcal{F}(\Gamma_{t-1}, I)} w(\mathcal{O}'(\Gamma_t, I^1_{\xi_{t+1}}) + \ldots + w(\mathcal{O}'(\Gamma_t, I^k_{\xi_{t+1}})) = \max_{\gamma_t \in \mathcal{F}(\Gamma_{t-1}, I)} f(\gamma_t),$$

where $I^k_{\xi_{t+1}}$ is the k-th sample of $I_{\xi_{t+1}}$ drawn from the underlying distribution. The sum shown on the left side of the upper equation is the exact sum that algorithm $\mathcal{E}$ computes in line 11 for each feasible request $j \in \mathcal{F}$. Algorithm $\mathcal{E}$ might have a poor performance if n is small since the feasible decisions are only evaluated on a small number of samples [1].

It is necessary to further investigate line 15 of algorithm $\mathcal{E}$. Since the iteration starting at this line is executed exactly $\lfloor n/|\mathcal{F}| \rfloor$ times, it is possible that no iteration at all is performed, if $|\mathcal{F}| > n$. Since the evaluation function is initialized with zero for each feasible request, no reasonable choice is made in this case. Setting n large enough will solve this issue, but since the number of feasible requests is not fixed, an alternative solution is necessary. Therefore, whenever ties in finding the maximum evaluation value occur, it is advised to schedule the packet with highest weight from these tied packets. In the described scenario this will lead to a greedy scheduling which is in fact a reasonable choice. In an online framework this is preferred since it leads to a higher immediate decision reward (in this case a higher packet weight) with equivalent predicted influence on the future. This approach will lead to an increase in anticipativity, as defined in Section 1.3, when the maximum of the evaluation function is not unique.

Another specific case to consider is $|\mathcal{F}| = 1$. It is immediately clear that

in this case the only reasonable option is to schedule the unique feasible request. Obviously, this can be done without sampling and performing an optimization over the n samples.

4.4.2 Consensus Algorithm

Consensus algorithms are usually used to analyse distributed systems and deal with the agreement of a group of agents [43]. The *consensus algorithm* $\mathcal{C}$ for our packet scheduling problem, shown in Figure 4.7, incorporates this idea but in this case no agents are present, rather the agreement is strived over the different sampled scenarios S_F. Every time a packet is scheduled in a scenario at the current time t it gets a vote through the agreement function f_a. The packet with the highest agreement is eventually scheduled.

Algorithm $\mathcal{C}(I, n)$
1: identical to algorithm $\mathcal{A}$ (see Figure 4.5) but SELECT-$\mathcal{C}$ is used in line 6

SELECT-$\mathcal{C}(\boldsymbol{\gamma}, R, t, n)$
2: $\Gamma_{t-1} \leftarrow \{\gamma_1, \ldots, \gamma_{t-1}\}$
3: $\mathcal{F} \leftarrow \{j \in R \setminus \boldsymbol{\gamma} \mid H(\Gamma_{t-1} \cup \{j\})\}$
4: **for** $j \in \mathcal{F}$ **do**
5: $\quad f(j) \leftarrow 0$
6: **end for**
7: **for** $i \leftarrow 1 \ldots n$ **do**
8: $\quad S_F \leftarrow \{R \cup \text{SAMPLE}(t, h_s)\}$
9: $\quad \boldsymbol{\gamma}^* \leftarrow \mathcal{O}'(\boldsymbol{\gamma}, S_F)$
10: $\quad f_a(\gamma_t^*) \leftarrow f_a(\gamma_t^*) + 1$
11: **end for**
12: **return** $\arg\max_{j \in \mathcal{F}}(f(j))$

Figure 4.7: Consensus Algorithm $\mathcal{C}$

The algorithm itself is very similar to algorithm $\mathcal{E}$ but it doesn't evaluate each feasible request against each scenario. It again starts by calculating the set of feasible requests $\mathcal{F}$ (line 3). Again, n is the number of performed optimization steps. Algorithm $\mathcal{C}$ samples exactly n scenarios (line 7 and 8) and performs an optimization only once for each scenario (line 9). In this case, instead of $n/|\mathcal{F}|$ as in the case of algorithm $\mathcal{E}$, n scenarios are sampled with the same amount of performed optimization steps. Each packet j, scheduled at γ_t^* in the respective scenario, is credited with a constant reward (line 10). All other packets scheduled in $\gamma_{t'}^*$ at time $t' \neq t$ receive no reward. The

request j to be scheduled at γ_t is defined by the packet with the highest reward $f_a(j)$ (line 12).

The consensus algorithm $\mathcal{C}$ is used when time is limited and not many optimization steps n can be performed. In this case algorithm $\mathcal{E}$ has poor performance and algorithm $\mathcal{C}$ is designed to address this problem. It outperforms algorithm $\mathcal{E}$ significantly when time is limited [44]. This comes from the fact that a small n needs to be chosen for limited time which reduces the number of iterations of the sampling and optimization procedure (Figure 4.6 line 7) performed by the expectation algorithm.

4.4.3 Regret Algorithm

The last online anticipative algorithm discussed in this work tries to combine the properties of both algorithm $\mathcal{E}$ and algorithm $\mathcal{C}$ and is shown in Figure 4.8. The *regret algorithm* $\mathcal{R}$ considers exactly n scenarios and still evaluates all feasible requests on each of the n scenarios.

Algorithm $\mathcal{R}(I, n)$

1: identical to algorithm $\mathcal{A}$ (see Figure 4.5) but SELECT-$\mathcal{R}$ is used in line 6

SELECT-$\mathcal{R}(\boldsymbol{\gamma}, R, t, n)$

2: $\Gamma_{t-1} \leftarrow \{\gamma_1, \ldots, \gamma_{t-1}\}$
3: $\mathcal{F} \leftarrow \{j \in R \setminus \boldsymbol{\gamma} \mid H(\Gamma_{t-1} \cup \{j\})\}$
4: **for** $j \in \mathcal{F}$ **do**
5: $\quad f(j) \leftarrow 0$
6: **end for**
7: **for** $i \leftarrow 1 \ldots n$ **do**
8: $\quad S_F \leftarrow \{R \cup \text{SAMPLE}(t, h_s)\}$
9: $\quad \boldsymbol{\gamma}^* \leftarrow \mathcal{O}'(\boldsymbol{\gamma}, S_F)$
10: $\quad f(\gamma_t^*) \leftarrow f(\gamma_t^*) + w(\gamma^*)$
11: $\quad$ **for** $j \in \mathcal{F} \setminus \{\gamma_t^*\}$ **do**
12: $\quad\quad \gamma_t \leftarrow j$
13: $\quad\quad f(j) \leftarrow f(j) + w(\boldsymbol{\gamma}^*) - \text{REGRET}(\boldsymbol{\gamma}, S_F, \boldsymbol{\gamma}^*)$
14: $\quad\quad \gamma_t \leftarrow null$
15: $\quad$ **end for**
16: **end for**
17: **return** $\arg\max_{j \in \mathcal{F}}(f(j))$

Figure 4.8: Regret Algorithm $\mathcal{R}$

The implementation is very similar to algorithm $\mathcal{C}$, the only difference is

the segment starting in line 11 of algorithm $\mathcal{R}$, inherited from algorithm $\mathcal{E}$. Similar to the consensus algorithm $\mathcal{C}$, the regret algorithm $\mathcal{R}$ calculates the optimal solution $\boldsymbol{\gamma}^*$ based on each of the n sampled scenarios (line 9). In order to incorporate suboptimal solutions $\tilde{\boldsymbol{\gamma}}$, and therefore all suboptimal packets $j \in \tilde{\gamma}_t$ scheduled at time t, it also evaluates each feasible request against each scenario (line 11). For that it uses the function REGRET, shown in Figure 4.9. The evaluation function f sums the weights of the solutions (lines 10 and 13), similar to algorithm $\mathcal{E}$. Again, also algorithm $\mathcal{R}$ chooses the packet with the highest evaluation function to be scheduled at time t (line 17).

Since time is still limited, the offline optimization algorithm $\mathcal{O}'$ can't be used to evaluate each feasible request against each scenario to calculate all suboptimal solutions. An efficient way to address this limitation is to approximate the weight of the suboptimal solutions $\tilde{\boldsymbol{\gamma}}$, based on the already calculated optimal solution $\boldsymbol{\gamma}^*$ (line 13). Such a problem is called a *suboptimality approximation problem* [45] and it can be solved by using an algorithm $\tilde{\mathcal{O}}$ which returns the following approximation for the weight of the suboptimal solution:

$$w(\mathcal{O}'(\Gamma_t, I)) \leq \alpha w(\tilde{\mathcal{O}}(\Gamma_t, I, \mathcal{O}'(\Gamma_{t-1}, I))). \tag{4.3}$$

Besides the constant approximation with factor α, the approximation algorithm $\tilde{\mathcal{O}}$ also needs to satisfy efficiency requirements, so that calling algorithm $\tilde{\mathcal{O}}$ up to $|\mathcal{F}|$ times does not exceed the runtime of a single optimization call.

In practice it is not necessary to calculate the suboptimality approximation since computing the *suboptimality approximation regret* is sufficient [1]. It is defined as an upper bound on the local loss L when scheduling a suboptimal request j at time t, that is,

$$L \equiv w(\mathcal{O}(\Gamma_{t-1}, I)) - w(\mathcal{O}(\Gamma_t, I)). \tag{4.4}$$

The function REGRET calculates this approximation bound, that is,

$$\text{REGRET}(\Gamma_t, I, \mathcal{O}(\Gamma_{t-1}, I)) \geq w(\mathcal{O}(\Gamma_{t-1}, I)) - w(\mathcal{O}(\Gamma_t, I)),$$

under the assumption that all other decisions at time $t' > t$ are optimal. Since we already know the optimal solution $\boldsymbol{\gamma}^*$ from the optimization step in line 9, we can compute the regret of the suboptimality approximation, that is

$$w(\mathcal{O}'(\Gamma_t, I)) - \text{REGRET}(\Gamma_t, I, \mathcal{O}(\Gamma_{t-1}, I)). \tag{4.5}$$

Equation 4.5 is implemented in line 13 of the regret algorithm $\mathcal{R}$ for evaluating the suboptimal feasible requests against each scenario.

This paragraph now discusses the algorithm REGRET, used to implement the calculation of the suboptimality approximation regret (Figure 4.9). The basic idea is to take the optimal solution $\boldsymbol{\gamma^*}$, schedule the suboptimal packet $\boldsymbol{\tilde{\gamma}_t}$ at time t and calculate the regret that is caused by this premise.

Algorithm REGRET($\boldsymbol{\tilde{\gamma}}, S_F, \boldsymbol{\gamma^*}$)

1: $R \leftarrow S_F \setminus \boldsymbol{\gamma^*}$
2: $t_c \leftarrow \max\{t \in T \mid \tilde{\gamma}_t \neq null\}$
3: **if** $\tilde{\gamma}_{t_c} \notin \boldsymbol{\gamma^*}$ **then**
4: **return** $\min\{w(\gamma^*_t) - w(\tilde{\gamma}_{t_c}) \mid t \in T \text{ and } t_c \leq t \leq a(\gamma^*_{t_c}) + d\}$
5: **else**
6: $t^*_c \leftarrow t \in T \mid \gamma^*_t = \tilde{\gamma}_{t_c}$
7: **if** $a(\gamma^*_{t_c}) \leq t^*_c \leq a(\gamma^*_{t_c}) + d$ **then**
8: **return** 0
9: **else**
10: $S \leftarrow \{t \in T \mid t_c + 1 \leq t \leq a(\gamma^*_{t_c}) + d \text{ and } w(\gamma^*_t) \leq w(\gamma^*_{t_c})\}$
11: **if** $S \neq \{\}$ **then**
12: $t_r \leftarrow \arg\min_{t \in S}(w(\gamma^*_t))$
13: $w_r \leftarrow \max\{w(r) \mid r \in R \cup \{\gamma^*_{t_r}\} \text{ and } a(r) \leq t^*_c \leq a(r) + d\}$
14: **return** $w(\gamma^*_{t_r}) - w_r$
15: **else**
16: $w_r \leftarrow \max\{w(r) \mid r \in R \text{ and } a(r) \leq t^*_c \leq a(r) + d\}$
17: **return** $w(\gamma^*_{t_c}) - w_r$
18: **end if**
19: **end if**
20: **end if**

Figure 4.9: Suboptimality Approximation Regret Calculation

The algorithm starts by defining the set of available requests, being the requests from the sampled scenario S_F without the packets already scheduled in the optimal solution $\boldsymbol{\gamma^*}$ (line 1). If the suboptimal packet $\tilde{\gamma}_{t_c}$, scheduled at the current time t_c, is not scheduled in the optimal solution $\boldsymbol{\gamma^*}$ (line 3), the algorithm tries to reschedule the optimal packet $\gamma^*_{t_c}$ to another time slot t, such that as little cost as possible is lost (line 4). Otherwise, if packet $\tilde{\gamma}_{t_c}$ is already scheduled in the optimal solution $\boldsymbol{\gamma^*}$ at some time t^*_c (meaning that $\tilde{\gamma}_{t_c} = \gamma^*_{t^*_c}$), it is first tried to swap those two packets in the optimal schedule $\boldsymbol{\gamma^*}$ (line 7) which would result in a cost-equivalent schedule with zero regret (line 8). If these packets cannot be exchanged in the optimal

schedule $\boldsymbol{\gamma^*}$, another packet needs to be found that can be scheduled at time t_c^* so that packet $\tilde{\gamma}_{t_c}$ is not scheduled twice. Algorithm REGRET first tries to reschedule $\gamma_{t_c}^*$ in place of another packet scheduled at time t_r, such that $w(\gamma_{t_r}^*) \leq w(\gamma_{t_c}^*)$ (lines 10 and 12). Then the best possible packet from the available requests is chosen to be scheduled at time t_c^* (line 14). Note that since packet $w(\gamma_{t_r}^*)$ is dropped, also this packet can be chosen to be scheduled at time t_c^*. At last, if no such packet $\gamma_{t_r}^*$ can be found, the algorithm simply schedules the highest weighted available packet at time t_c^* (line 17) and the optimal packet $\gamma_{t_c}^*$ is lost.

This implementation covers all possible scenarios so it will always return a regret that corresponds to a valid suboptimal scheduling that has been created from the optimal solution $\boldsymbol{\gamma^*}$. Algorithm REGRET takes sublinear time and is therefore computationally negligible for this application [1].

4.4.4 Q-Learning Algorithm

This section now discusses the implementation of the classical *Q-learning algorithm* $\mathcal{RL}$, shown in Figure 4.10. It starts by defining the set of feasible requests $\mathcal{F}$ (line 3), which are then used to update the current state s_t. This is done in line 5.

Algorithm $\mathcal{RL}(I)$

1: identical to algorithm $\mathcal{A}$ (see Figure 4.5) but SELECT-$\mathcal{RL}$ is used in line 6

SELECT-$\mathcal{RL}(\boldsymbol{\gamma}, R, t, Q, N, \tilde{r}_t, a_t, s_t)$

2: $\Gamma_{t-1} \leftarrow \{\gamma_1, \ldots, \gamma_{t-1}\}$
3: $\mathcal{F} \leftarrow \{j \in R \setminus \boldsymbol{\gamma} \mid H(\Gamma_{t-1} \cup \{j\})\}$
4: $s_{t-1} \leftarrow s_t$
5: $s_t \leftarrow$ observe the new state
6: $f \leftarrow 1/(1 + N(s_{t-1}, a_t))$
7: $Q(s_{t-1}, a_t) \leftarrow (1 - f)\, Q(s_{t-1}, a_t) + f(\tilde{r}_t + \rho \max_{a \in A} Q(s_t, a))$
8: $N(s_{t-1}, a_t) \leftarrow N(s_{t-1}, a_t) + 1$
9: $a_t \leftarrow \arg\max_{a \in A}\{Q(s_t, a) \mid a = w(r) \text{ and } r \in \mathcal{F}\}$
10: $\gamma_t \leftarrow \arg\min_{r \in \mathcal{F}}\{a(r) \mid w(r) = a_t\}$
11: $\tilde{r}_t \leftarrow w(a_t)$

Figure 4.10: Q-Learning Algorithm $\mathcal{RL}$

Afterwards, the Q-function is updated for the action chosen at time $t - 1$, together with the function N (line 8). The function N keeps track on how many times each state-action pair has been visited so far. The action a_t is

then chosen by the maximum Q-value for the state s_t. Note, that rather an exploratory strategy should be used here as defined in Section 3.3, but for simplicity such variants are not included in the implementation. The packet r to be scheduled at time t is the feasible packet with earliest arrival time $a(t)$, given its weight $w(r)$ matches the chosen action a_t. Therefore, the action a always corresponds to the weight of a specific packet class. Note that the functions Q and N need to be initialized at the start, together with the initial action $a_0 = 0$. For the function Q an optimistic initialisation is advised to encourage early exploration and avoid getting stuck on suboptimal actions. This means that the initialisation of each Q-entry is set higher than its true value by using an appropriate upper bound [46].

Still left to define is the state representation for the packet scheduling problem. The proposed approach in this work is to define a state as a vector of integer values, where each value describes the currently available packet quantity of a specific packet class C_w and the time left until expiration, according to Table 4.1. This is based on the idea that the packet with earliest arrival is most interesting to be described in more detail for each class. For such a packet its main feature is the time until it expires.

Code	Description
0	$C_w = \{\}$
1	$\lvert C_w \rvert = 1$ and $t - \min_{r \in C_w} a(r) + d = 0$
2	$\lvert C_w \rvert \geq 1$ and $t - \min_{r \in C_w} a(r) + d = 0$
3	$\lvert C_w \rvert = 1$ and $t - \min_{r \in C_w} a(r) + d = 1$
4	$\lvert C_w \rvert \geq 1$ and $t - \min_{r \in C_w} a(r) + d = 1$
5	$\lvert C_w \rvert = 1$ and $t - \min_{r \in C_w} a(r) + d > 1$
6	$\lvert C_w \rvert \geq 1$ and $t - \min_{r \in C_w} a(r) + d > 1$

Table 4.1: State Encoding Scheme for the Reinforcement Learning

The vector created according to the described scheme can then easily be converted to an integer and be used for hashing the Q-table. However, it needs to be ensured that the structure used to represent the Q-function is chosen large enough, such that the index is always in the desired range.

4.4.5 Conservative Q-Learning Algorithm

With a slightly different approach to the classical Q-learning algorithm, now the *conservative Q-learning algorithm* $\mathcal{RLC}$ is presented in Figure 4.11. The start of the algorithm is equivalent to the previously defined algorithm $\mathcal{RL}$.

Also the state representation and the state updates remain unchanged. However, note the slightly different update rule of the Q-function in line 8. Now the maximum weight w_{max} from the available packets is used as a lower bound on the Q-update. The packet to be scheduled itself is again chosen identical to the classical Q-learning algorithm, but the remaining part of the algorithm $\mathcal{RLC}$ is now significantly different.

Algorithm $\mathcal{RLC}(I)$

1: identical to algorithm $\mathcal{A}$ (see Figure 4.5) but SELECT-$\mathcal{RLC}$ is used in line 6

SELECT-$\mathcal{RLC}(\boldsymbol{\gamma}, R, t, Q, N, \tilde{r}_t, a_t, s_t, w_{max})$

2: $\Gamma_{t-1} \leftarrow \{\gamma_1, \ldots, \gamma_{t-1}\}$
3: $\mathcal{F} \leftarrow \{j \in R \setminus \boldsymbol{\gamma} \mid H(\Gamma_{t-1} \cup \{j\})\}$
4: $w_{max} \leftarrow \max_{r \in \mathcal{F}} w(r)$
5: $s_{t-1} \leftarrow s_t$
6: $s_t \leftarrow$ observe new state
7: $f \leftarrow 1/(1 + N(s_{t-1}, a_t))$
8: $Q(s_{t-1}, a_t) \leftarrow (1 - f)\, Q(s_{t-1}, a_t) + f(\tilde{r}_t + \rho\, w_{max})$
9: $N(s_{t-1}, a_t) \leftarrow N(s_{t-1}, a_t) + 1$
10: $a_t \leftarrow \arg\max_{a \in A}\{Q(s_t, a) \mid a = w(r) \text{ and } r \in \mathcal{F}\}$
11: $\gamma_t \leftarrow \arg\min_{r \in \mathcal{F}}\{a(r) \mid w(r) = a_t\}$
12: $L = \{r \in \mathcal{F} \setminus \{\gamma_t\} \mid a(r) + d = t\}$
13: $l_t \leftarrow \sum_{r \in L} w(r)$
14: $\tilde{r}_t \leftarrow w(a_t) - l_t$

Figure 4.11: Conservative Q-Learning Algorithm $\mathcal{RLC}$

First, the set L of packets that will be lost at the next time step is calculated in line 12. This is used to calculate the explicit loss (line 13) which is then used to reduce the reward function $\tilde{r}$.

4.4.6 Q-Learning Algorithm with Belief States

This section now applies the approach using belief states discussed in Section 3.4.2 to the packet scheduling problem, by defining the *Q-learning Algorithm with Belief States* $\mathcal{RLB}$. The implementation of the algorithm itself is almost identical to the classical Q-learning algorithm $\mathcal{RL}$ and therefore not listed again. The only difference lies in the update of the states, that now is a function of the entire observed history. Since now the states must encode the traffic of each packet class together with the belief state of the distribution of

each packet class, it is advised to represent the states as two vectors of integer values. The first is equivalent to the representation discussed previously in Section 4.4.4 and the second vector contains the information on the belief states, again one integer value for each packet class. When converting these vectors into integer values, now two hash values are available. This however is no limitation since the Q-function can easily be defined as a multidimensional table. The only limitation here is the heap space that needs to be considered, since the tables grow with the amount of different belief states and the quantity of distinct packet classes.

The exact implementation of the procedure for updating the states s_t is not shown in detail since it varies for each type of distribution the traffic is modelled of.

Chapter 5

Learning Input Distributions

Previously in this work when discussing online anticipative algorithms, full knowledge of the input distribution $\mathcal{I}$ has always been assumed. In practice however this is rarely the case since either no information at all, or only partial information is available for the underlying input distribution $\mathcal{I}$. Still, having a distribution to sample from is absolutely necessary to apply the algorithmic framework discussed in Chapter 2. A reasonable approach is to use the existing information of the past to learn the distribution $\mathcal{I}$. There might be already data available or the learning process needs to be performed entirely during the online optimization effort.

In general the two main approaches to such a learning task are *statistical modelling* and *machine learning*. Statistics emphasizes more on the aspect of drawing population inferences from a sample, whereas the aim of machine learning is to find generalizable predictive patterns. Furthermore, learning statistical models requires some existing knowledge of the system to predefine the model to be learned. Machine learning for the most part only requires choosing the predictive algorithm [47].

The first section of this chapter discusses the case when some information of the underlying distribution $\mathcal{I}$ is available. A hidden Markov model is assumed, where transition probabilities are not known a priori and therefore have to be estimated over time using a statistical approach. The second and third section discuss the case where nothing at all is known about the underlying distribution $\mathcal{I}$ and machine learning techniques are applied to learn the model over the time horizon. Either approach is applied to the online setting, where inputs are revealed over time. Therefore, information on the distribution $\mathcal{I}$ is revealed step by step and can be used to infer the state of the distribution or to train the partially defined model [1]. For the implementation this requires to extend the generic online algorithm $\mathcal{A}$ to incorporate an additional learning step at each time step t. This learning

step LEARN uses the information of the request set R_t, revealed at time t, for continuous improvements of the existing predictive model. Of course, the actual implementation of this step depends on the chosen predictive model, but generally this step adjusts the predictor's parameters to the new observed data. The generic algorithm $\mathcal{A}'$ is shown in Figure 5.1 below.

Algorithm $\mathcal{A}'(I)$
1: **for** $t \in T$ **do**
2: $\quad \gamma_t \leftarrow null$
3: **end for**
4: **for** $t \in T$ **do**
5: $\quad R \leftarrow \cup_{R_{t'} \in I, t' \leq t} R_{t'}$
6: $\quad \tilde{I} \leftarrow$ LEARN(R)
7: $\quad \gamma_t \leftarrow$ SELECT$(\boldsymbol{\gamma}, R, t, \tilde{I})$
8: **end for**
9: **return** $\boldsymbol{\gamma}$

Figure 5.1: Generic Online Algorithm $\mathcal{A}'$ with Learning

As time passes the quality of the predictive model is expected to improve and therefore also the quality of the function SAMPLE should yield better and better results throughout the online optimization effort. This effect is investigated in Chapter 6.

5.1 Hidden Markov Models

In this chapter the underlying distribution $\mathcal{I}$ is assumed to be a set of *Markov models*, for the reason that a Markov model is complex enough to describe a variety of real-world time series [48]. Each Markov model in $\mathcal{I}$ describes the arrival of a packet class, determined by its weight. A request at a specific time t in the online framework can then be composed of the packets emitted when performing a transition in each individual Markov model.

A Markov model can be described by using five attributes. Each model Λ has a set of states S, a set of possible outputs O, a transition probability function p, an output probability function b and an initial state distribution μ at time $t = 1$ [49]. The transition probability function $p(s_t, s_{t+1})$ describes the probability that the current state s_t is changed to the state s_{t+1}, while the transmitted output o_{t+1} in the new state is described by $b(s_{t+1}, o_{t+1})$. For *hidden Markov models* the current state of the distribution and the transition

and output probabilities are not observable. In this way, they are models with incomplete data that comprise unobserved random variables.

Since the state $s_t \in S$ of the input distribution $\mathcal{I}$ is not known in the hidden Markov model, the information on the past sequence of outputs needs to be used to infer the current state. Therefore, the first problem to solve is finding a state sequence $\boldsymbol{s}$, that best explains the observed output sequence $\boldsymbol{o} = (o_1, \ldots, o_\tau)$ given a specific model Λ, where τ is the current time step of the online optimization procedure. This optimal state sequence is not unique since several optimality criteria can be chosen. In this work the *individually* most likely states s_t are chosen. This can be done by introducing the variable

$$\phi(s) = P(s_t = s \mid \boldsymbol{o}, \Lambda),$$

which describes the probability of being in state s at time t, given the model and the observations so far. In order to calculate ϕ, two algorithms need to be introduced first. These algorithms constitute the core of the *Baum-Welch method* [50], shown in Figure 5.2 below, that is used to adjust the parameters of the hidden Markov model at each time step τ of the online optimization effort.

5.1.1 Forward and Backward Algorithm

In the following the observation sequence $\boldsymbol{o}$ and an initial state s_1 are fixed. The *forward algorithm* calculates the forward variable

$$\alpha_t(s) = P((o_1, \ldots, o_t), s_t = s \mid \Lambda),$$

that is, the probability of being in state s at time $t < \tau$, when starting from state s_1 and taking into consideration the observations $(o_1, \ldots, o_t)$. The calculation of the forward variable is recursively defined as

$$\alpha_{t+1}(s) = \left(\sum_{s' \in S} \alpha_t(s) p(s', s) \right) b(s, o_{t+1}), \tag{5.1}$$

with the initialisation

$$\alpha_1(s) = \mu(s) b(s, o_1). \tag{5.2}$$

Since $\alpha_t(s)$ is the probability that $(o_1, \ldots, o_t)$ is observed with eventually reaching the final state $s_t = s$, $\alpha_\tau(s)$ is the probability of the entire observation sequence $\boldsymbol{o}$ given the last state $s_\tau = s$. It is therefore easy to see that

the probability $P(\boldsymbol{o} \mid \Lambda)$, the probability of the entire output sequence given the model, is the sum

$$P(\boldsymbol{o} \mid \Lambda) = \sum_{s \in S} \alpha_\tau(s). \tag{5.3}$$

The property shown above in equation 5.3 will be important for the formulation of a possible stopping criterion.

In a similar way the *backward algorithm* can be defined. It calculates the backward variable

$$\beta_t(s) = P((o_{t+1}, \ldots, o_\tau) \mid s_t = s, \Lambda),$$

that is, the probability that $(o_{t+1}, \ldots, o_\tau)$ is observed when starting from state s at time t and given the model Λ. Here the recursion is defined as

$$\beta_t(s) = \sum_{s' \in S} p(s, s') b(s', o_{t+1}) \beta_{t+1}(s'), \tag{5.4}$$

with the initialisation

$$\beta_\tau(s) = 1. \tag{5.5}$$

Now we can express $\phi_t(s)$ in terms of the backward and forward variables using the simple relation

$$\phi_t(s) = \frac{\alpha_t(s)\beta_t(s)}{\sum_{s' \in S} \alpha_t(s')\beta_t(s')}. \tag{5.6}$$

Having the first problem solved, $\phi_t(s)$ can now be used to estimate the most probable current state s_t for each time step. The immediate problem following is how to describe the other unobserved random variables, the transition and output probabilities.

5.1.2 Baum-Welch Algorithm

The second question to solve is how the model parameters mu, p and b need to be adjusted in order to maximise the probability $P(\boldsymbol{o} \mid \Lambda)$. There is actually no optimal way of estimating the model parameters, but the iterative Baum-Welch method can be used to locally maximise the probability $P(\boldsymbol{o} \mid \Lambda)$ [51]. In order to be able to formulate the re-estimation of the model parameters one more probability needs to be introduced. The variable

$$\xi(s, s') = P(s_t = s, s_{t+1} = s' \mid \boldsymbol{o}, \Lambda)$$

expresses the probability of being in state s at time t and in state $s' \in S$ at time $t+1$, again given the model and the observed output sequence. Also for the calculation of this variable we will use the forward and backward variables that have already been calculated and write it as

$$\xi_t(s, s') = \frac{\alpha_t(s)p(s, s')b(s', o_{t+1})\beta_{t+1}(s')}{\sum_{s'' \in S} \sum_{s''' \in S} \alpha_t(s'')p(s'', s''')b(s''', o_{t+1})\beta_{t+1}(s''')}. \tag{5.7}$$

In order to re-estimate the model parameters, the properties of the calculated probabilities $\phi_t(s)$ and $\xi_t(s, s')$ need to be further investigated. Now, these variables are summed over the time horizon $[1, \ldots, \tau - 1]$ for each state $s \in S$ individually. But even before that, a simple re-estimation of the initial state distributions can be given by

$$\bar{\mu}_s = \phi_1(s). \tag{5.8}$$

The summation of $\phi_t(s)$ over time will then give an estimate on the number of times the state s is visited, or more precisely the number of transitions made from s. In a similar manner the summation of the variable $\xi_t(s, s')$ over time can be interpreted as the number of transitions from state s to state s'. This knowledge on the transitions will allow us to re-estimate the transition probabilities

$$\bar{p}(s, s') = \frac{\sum_{t \in [1, \ldots, \tau-1]} \xi_t(s, s')}{\sum_{t \in [1, \ldots, \tau-1]} \phi_t(s)}. \tag{5.9}$$

For the re-estimation of the output probabilities a similar equation is used, but in the nominator the summation of $\phi_t(s)$ over time is done only when the output at that time corresponds to the output probability to be estimated. This can be written as

$$\bar{b}(s, o) = \frac{\sum_{t \in [1, \ldots, \tau-1] | o_t = o} \phi_t(s)}{\sum_{t \in [1, \ldots, \tau-1]} \phi_t(s)}, \tag{5.10}$$

which corresponds to a re-estimation according to the actual output frequency of each possible output $o \in O$ for each state.

Since the equations defined above are an iterative process, a stopping criterion needs to be defined. As already mentioned previously in this chapter, the probability $P(\boldsymbol{o} \mid \Lambda)$ is used to formulate this criterion. More precisely, the iterative procedure is continued until the probability $P(\boldsymbol{o} \mid \Lambda)$ does not change much, or until

$$\left| P(\boldsymbol{o} \mid \Lambda) - P(\boldsymbol{o} \mid \bar{\Lambda}) \right| \leq \epsilon, \tag{5.11}$$

where $\bar{\Lambda}$ are the re-estimated model parameters and ϵ is a precision parameter.

An implementation of the function LEARN for hidden Markov models using the Baum-Welch algorithm is shown below in Figure 5.2. The presented implementation is suitable to be applied in an online framework where the model parameters are optimized for each time step in the past horizon $[t-\lambda,\dots,t]$.

Algorithm LEARN-HMM$(\boldsymbol{o}, t, \lambda, \bar{p}, \bar{b}, \phi)$

1: $t_s \leftarrow \max(0, t-\lambda)$
2: **for** $s \in S$ **do**
3: $\quad \alpha_{t_s}(s) \leftarrow \phi_{t_s}(s)\bar{b}(s, o_{t_s})$
4: $\quad \beta_{t_s}(s) \leftarrow 1$
5: **end for**
6: **repeat**
7: $\quad p_0 \leftarrow P((o_{t_s},\dots,o_t))$
8: $\quad$ **for** $t' \leftarrow t_s \dots t-1$ **do**
9: $\quad\quad$ **for** $s \in S$ **do**
10: $\quad\quad\quad \alpha_{t'+1}(s) \leftarrow \left(\sum_{s' \in S} \alpha_{t'}(s)\bar{p}(s', s)\right) \bar{b}(s, o_{t'+1})$
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: $\quad$ **for** $t' \leftarrow t-1 \dots t_s$ **do**
14: $\quad\quad$ **for** $s \in S$ **do**
15: $\quad\quad\quad \beta_{t'}(s) \leftarrow \sum_{s' \in S} \bar{p}(s, s')\bar{b}(s', o_{t'+1})\beta_{t'+1}(s')$
16: $\quad\quad$ **end for**
17: $\quad$ **end for**
18: $\quad$ **for** $t' \leftarrow t_s \dots t-1$ **do**
19: $\quad\quad \forall(s \in S, s' \in S)$: calculate $\xi_{t'}(s, s')$ using equation 5.7
20: $\quad\quad \forall s \in S$: calculate $\phi_{t'}(s)$ using equation 5.6
21: $\quad$ **end for**
22: $\quad$ **for** $s \in S$ **do**
23: $\quad\quad \phi_t(s) \leftarrow \alpha_t(s) / \sum_{s' \in S} \alpha_t(s')$
24: $\quad$ **end for**
25: $\quad$ **for** $s \in S$ **do**
26: $\quad\quad \forall o \in O$: re-estimate $b(s, o)$ using equation 5.10
27: $\quad\quad \forall s' \in S$: re-estimate $p(s, s')$ using equation 5.9
28: $\quad$ **end for**
29: **until** $|P((o_{t_s},\dots,o_t)) - p_o| \leq \epsilon$

Figure 5.2: Algorithm for Learning Hidden Markov Models

Of course, if $t-\lambda \leq 0$, the past horizon interval starts from time $t = 1$ (line 1). Note that in the implementation the model parameters $(\bar{p}, \bar{b}, \phi) \in \Lambda$ are given to the function to be iteratively re-estimated. Starting from line 2 the base cases of the forward and backward variables are initialised. Then, first the forward algorithm starting in line 8 and afterwards the backward algorithm starting in line 13 are implemented. Then the probability variables $\phi_t(s)$ and $\xi_t(s, s')$ are calculated (lines 18 to 24) and immediately afterwards the parameters of the hidden Markov model are re-estimated starting in line 25. This iteration is proceeded until the stopping criterion in line 29 is met. For stability and runtime guarantee it is advisable to restrict the optimization with an additional constraint on the number of maximum iterations.

At the start of the online optimization procedure $\tau = 1$, an initial estimation of the model parameters needs to be given to the algorithm since no previous result is available. In general, it is not possible to define an initial estimation that will certainly lead to a good local maximum of the Baum-Welch algorithm. Experimental results have shown that either random or uniform initial estimates can be used. However, good initial estimates for the probability function $b(s, o)$ can be helpful [51]. Recall that our framework is an online problem. By that, the Baum-Welch algorithm is executed in each time step and we can accept the model parameters adjusted at time t as initial values for time $t+1$. In that manner the actual initialisation will only affect the first execution of the Baum-Welch algorithm where not much information is available yet. Therefore, initial estimations in an online framework might not significantly influence the convergence. However, note that attention needs to be paid to the case when some of the model parameters are estimated to be zero. If that occurs, the zero estimated parameter will remain at zero also after the re-estimation procedure [51]. At the start of the online optimization where only little data is accessible this might happen frequently and a simple solution is to set the zero estimated probabilities to a fixed small value.

5.1.3 Precision Range and Scaling

When calculating the forward variables using their recursive definitions, the computation of the sum of many terms of the form

$$\prod_{t \in [1,\dots,\tau-1]} p(s_t, s_{t+1}) \prod_{t \in [1,\dots,\tau]} b(s_t, o_t)$$

is necessary. Since both terms are products of probabilities, that furthermore are generally significantly less than one, each term exponentially heads to

zero. If the time horizon τ is large (e.g. $\tau > 100$) the computation of the forward variables will exceed the precision range of any machine, even in double precision [51]. This of course also applies to the backward variable that is calculated using a similar recursion. Therefore, scaling needs to be incorporated in order to keep the variables within the precision range of the machine.

The idea is to scale each $\alpha_t(s)$ with a scaling factor c_t, and then the scaled forward variable $\bar{\alpha}_t(s)$ is used to proceed the recursion. The scaling factor c_t is chosen such that the sum of $\bar{\alpha}_t(s)$ at each time t is exactly 1, therefore

$$c_t = \frac{1}{\sum_{s \in S} \alpha_t(s)}. \tag{5.12}$$

The introduction of the scaled forward variable also slightly alters the recursion defined in equation 5.1 and can now be written as

$$\alpha_t(s) = \left(\sum_{s' \in S} \bar{\alpha}_{t-1}(s) p(s', s) \right) b(s, o_t). \tag{5.13}$$

Since the recursion only considers scaled forward variables it is also necessary to compute

$$\bar{\alpha}_t(s) = c_t \alpha_t(s) = \frac{\alpha_t(s)}{\sum_{s' \in S} \alpha_t(s')}. \tag{5.14}$$

It has already been mentioned that scaling also needs to be applied to the backward variable. Since the magnitude of both variables is comparable, the same scaling factor c_t can be used to scale the backward variable at each time step t, such that

$$\bar{\beta}_t(s) = c_t \beta_t(s). \tag{5.15}$$

Note that now the computation of $\xi_t(s, s')$ and $\gamma_t(s)$ also needs to be expressed in terms of the scaled forward and backward variables. It can be easily shown that multiplication with a term only depending on time (not the possible states) does not alter the actual result of equation 5.6 and 5.7, since the scaling factors cancel each other out [51]. For better readability, the actual computational steps of the scaling within the Baum-Welch algorithm are not included in the implementation shown in Figure 5.2.

5.2 Historical Averaging

This and the following sections now discuss the relaxation of the assumption on the presence of any information on the underlying input distribution $\mathcal{I}$.

This means that for the predictive mechanism only the already revealed input sequences, also denoted the *historical data*, are available.

A very simple approach is *historical averaging* shown in Figure 5.3. The idea is to observe the latest λ elements of the input sequence $\boldsymbol{o}$ and estimate the arrival probability of each $o \in O$ by simply considering their frequencies [1]. That means that the occurrence of each possible output $o \in O$ in these recently observed requests is counted to estimate the probability $P(o)$. Then, for the time horizon to be sampled, the sequence $(o_{t+1}, \ldots, o_{t+h_s})$ is generated by drawing a random request $o \in O$ with probability $P(o)$ at each time step.

Algorithm LEARN-HA$(\boldsymbol{o}, t)$

1: **for** $o \in O$ **do**
2: $\quad C(o) \leftarrow 0$
3: **end for**
4: **for** $i \leftarrow t - \lambda + 1 \ldots t$ **do**
5: $\quad C(o_i) \leftarrow C(o_i) + 1$
6: **end for**
7: **for** $o \in O$ **do**
8: $\quad P(o) \leftarrow \frac{C(o)}{\lambda}$
9: **end for**

Figure 5.3: Implementation of Historical Averaging

Incorporating only the latest λ elements is important, since the distribution $\mathcal{I}$ is not expected to be independent and identically distributed. The expected structure and temporal correlation of the data can be a reason why averaging over the entire time horizon is not feasible.

5.3 Historical Sampling

If it is expected that the underlying distribution $\mathcal{I}$ inherits some sort of structure, historical averaging might not be suitable because the structure of the inputs is entirely ignored. An approach to incorporate structural sequences in the data is *historical sampling*, presented in Figure 5.4. This approach is especially useful if the distribution $\mathcal{I}$ can be described using a Markov model, since in this case historical sampling represents a random walk in the model. Of course, this approach also can be applied if the underlying input distribution cannot be characterized by a Markov model. The basic idea is to select a past instance and use the output sequence starting with

the same instance also as outputs for the time interval to be sampled. The function RANDOM($[a, b]$) draws a random uniformly distributed value in the specified interval. There is actually no learning procedure to be implemented in this case, since the inputs are directly sampled from the past. The start of the chosen past sequence t is random (line 1) and, starting from this point, the sequence of outputs $(o_{t'}, \ldots, o_{t'+h_s})$ is taken and used as possible future outputs.

Algorithm SAMPLE-HS($\boldsymbol{o}, t, h_s$)
1: $t' \leftarrow$ RANDOM($[0, t - h_s]$)
2: **return** $(o_{t'+1}, \ldots, o_{t'+h_s})$

Figure 5.4: Implementation of Historical Sampling

A special case arises if $t - h_s < 0$. In this case not enough past information is available to be able to sample enough requests. There are several approaches to handle this situation, one being the iterative concatenation of the observed past until the sequence contains h_s requests. Certainly, this is not optimal, but at least it allows to extract any information out of the recent past.

5.4 Machine Learning

A general approach to predict future inputs is *machine learning*. It includes all computational predicting methods that use only the information on the past to improve their performance, that is in this case the observed past input sequences. Many machine learning tools are generically deterministic, meaning that the result is a specific prediction that can be repeated any time if the model parameters and the input remain unchanged. However, recall that the basic concept of the anticipatory algorithms discussed in Chapter 2, is the optimization in different sampled possible scenarios. If the prediction is deterministic, there is only one scenario that can be obtained, hence optimizing only in this scenario is possible. So the correct approach here would be to maximise the prediction accuracy and then just optimize in this predicted deterministic scenario. However, note that from many of these deterministic prediction tools a stochastic model can be obtained, e.g. by considering the activation percentage of an output node in neural networks.

Another point to consider is that the performance of machine learning tools strongly depends on the amount and quality of the training data [52]. Therefore, it might be necessary to have data accessible for pre-training, or otherwise it might take considerable time until the predictor is able to

make reasonable statements about the future in the online framework. In this framework, the machine learning algorithm is facing an *online learning* environment, where the algorithm receives a training example in each time step, makes a prediction, and then acquires a loss. The aim of the predictive model is to train its parameters to minimize the cumulative loss over all time steps t. Many online learning algorithms are used for adversarial scenarios but they can also be used to derive accurate predictors for a distributional scenario [53].

The machine learning model used in the context of online anticipatory algorithms is however not just confronted with a generic classification problem, rather it needs to face a *time series analysis* problem. Such an analysis requires the extraction of statistical information from data arranged in chronological order. Time series are interesting in this framework since they can address causality and trends, and therefore better forecast the future inputs. In time series data, the requests near in time are expected to be strongly correlated with one another, but an overall aggregation of the information without incorporation of the temporal correlations is usually not efficient for predicting the future [54].

A suitable machine learning model for the framework described above are *deep neural networks*, e.g. *convolutional neural networks* or *multilayer perceptrons*. Deep learning is highly flexible and complex enough to be advantageous for time series analysis. However, for the packet scheduling problem discussed in this work, it might be necessary to use a distinct deep network to model the arrival of each packet class separately.

Chapter 6

Experimental Analysis

The experimental analysis presented in this chapter was made to evaluate the different online anticipatory algorithms on the packet scheduling problem. The analysis is based on studying the average regret L defined in Section 1.1.2. To give more information on the actual behaviour of the algorithms, the cumulative average regret $L(t)$ at time t is considered. This might give more insight on an initial learning curve or other details compared to just analysing the overall regret within the time horizon. The function of the time dependent average regret $L(t)$ can be written as

$$L(t) = \mathbb{E}_I \left[\sum_{i \leq t} \left(w(\gamma_i^*) - w(\gamma_i) \right) \right],$$

where $\boldsymbol{\gamma}^*$ is the optimal decision sequence produced by the optimal offline algorithm $\mathcal{O}$ and $\boldsymbol{\gamma}$ is the decision sequence produced by the respective algorithm to be analysed. Note that at the end of the time horizon $t = h$ this value is equivalent to the definition of L given in equation 1.2.

Of course, due to the stochastic packet arrival the actual regret function is different for each instance of the problem, but still the overall drift can be noticed when averaging over several input sequences. For this analysis, the average is taken over five different instances. Note that each algorithm is applied on the same input sequences. Further, to remove some of the stochastic noise the average regret $L(t)$ is smoothed using a Gaussian-weighted moving average filter.

First, the greedy algorithm and the local optimization algorithm are compared to find the better suited oblivious approach for the packet scheduling problem. This approach will be then used in the following as a reference when evaluating the different online anticipatory algorithms.

6.1 Experimental Setting

This work considers the request arrival, that is, the underlying distribution $\mathcal{I}$, to be a set of hidden Markov models. This setting has already been used in previous work [1, 9]. Each hidden Markov model $\Lambda \in \mathcal{I}$ then describes the arrival of an individual packet class, characterized by its distinct weight. The randomly generated input distribution $\mathcal{I}$ that has been used for each experiment, is shown in Figure 6.1.

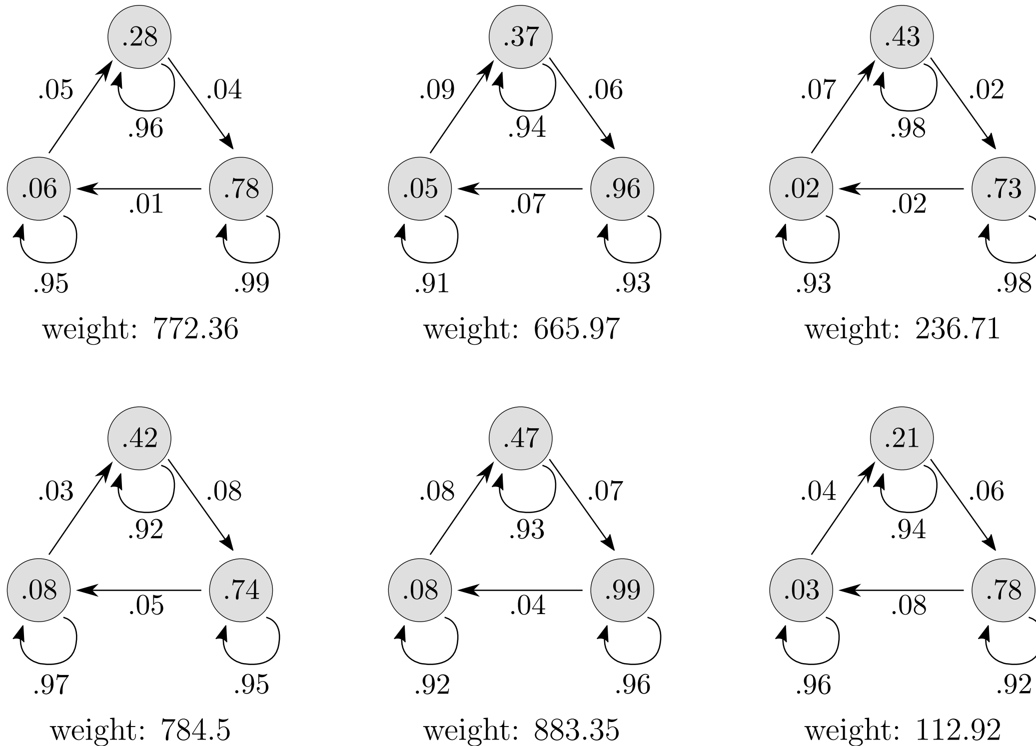


Figure 6.1: Experimental Model of the Packet Arrival

The hidden Markov models are created based on the previous research in [44]. All Markov models have three different states, corresponding to a low, medium and high request arrival rate. Each state has a dominating self-transition probability, drawn from the interval $[0.9, 1.0)$. Further a state can only change, apart from the self-transition, from either low to medium, medium to high or high to low. The probability that a request is emitted in a specific state of the hidden Markov model is drawn from the interval $(0, 0.1]$ for the low state, $[0.2, 0.5]$ for the medium state, and $[0.7, 1.0)$ for the high state. The deadline for all packets has been set to $d = 20$ and problems with up to six different packet classes are considered. The weights of the

packets are also drawn randomly from the interval $[0, 1000]$. An instance of an n-class problem always includes the first n classes (from left to right) shown in Figure 6.1. To generate a specific input sequence, for each time step every Markov model is sampled and if a packet is emitted, it is added to the request set at the respective time step.

6.2 Oblivious Algorithms

In the following, the oblivious online algorithms presented in Section 4.3 will be compared: the greedy algorithm $\mathcal{G}$ and the local optimization algorithm $\mathcal{L}$.

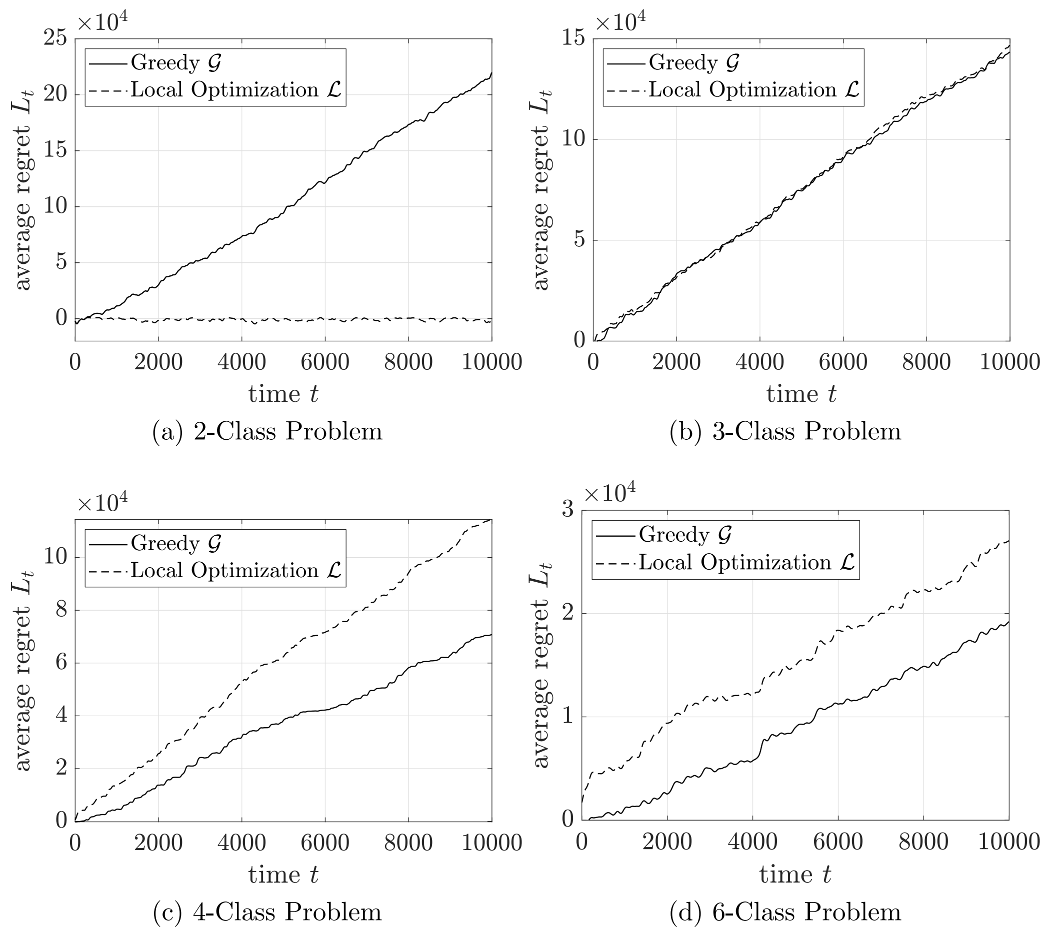
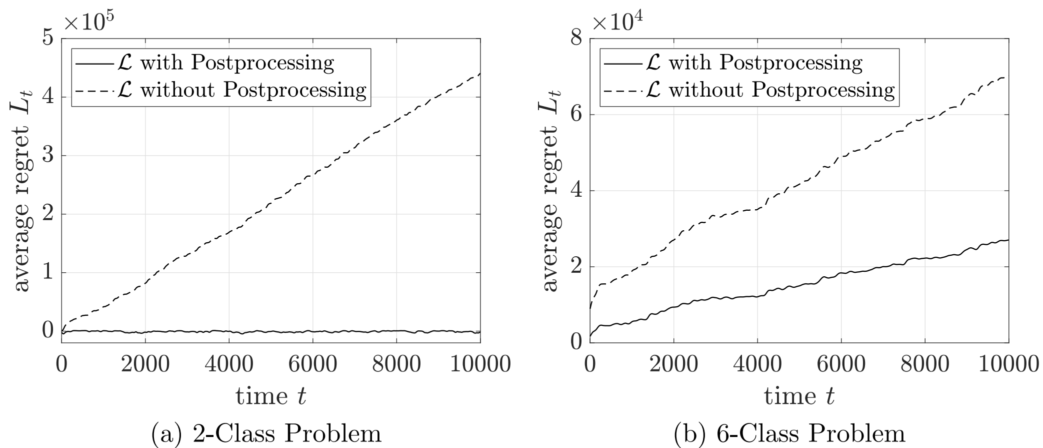


Figure 6.2: Comparison of Greedy $\mathcal{G}$ and Local Optimization $\mathcal{L}$

For the general n-class problem the greedy algorithm is the better approach. This is shown in Figure 6.2. For the special case that only two packet classes

are arriving, algorithm $\mathcal{L}$ has shown to be optimal in the experiments. With increasing class quantity however, the typical behaviour of over-optimization in online settings due to the absence of information on future requests [1] can be observed and greedy is a much better approach. The optimality of the local optimization algorithm is not extensively discussed in this work.

Remember, the local optimization algorithm executes the offline optimization algorithm $\mathcal{O}'$ at every time step. A postprocessing step has been defined for the optimal offline algorithm in Section 4.2.1. This step schedules high-weighted packets early and therefore improves the optimal solution in the online context. Since it also schedules the packets with shortest deadline early, this might be an optimal policy for the 2-packet case.



(a) 2-Class Problem (b) 6-Class Problem

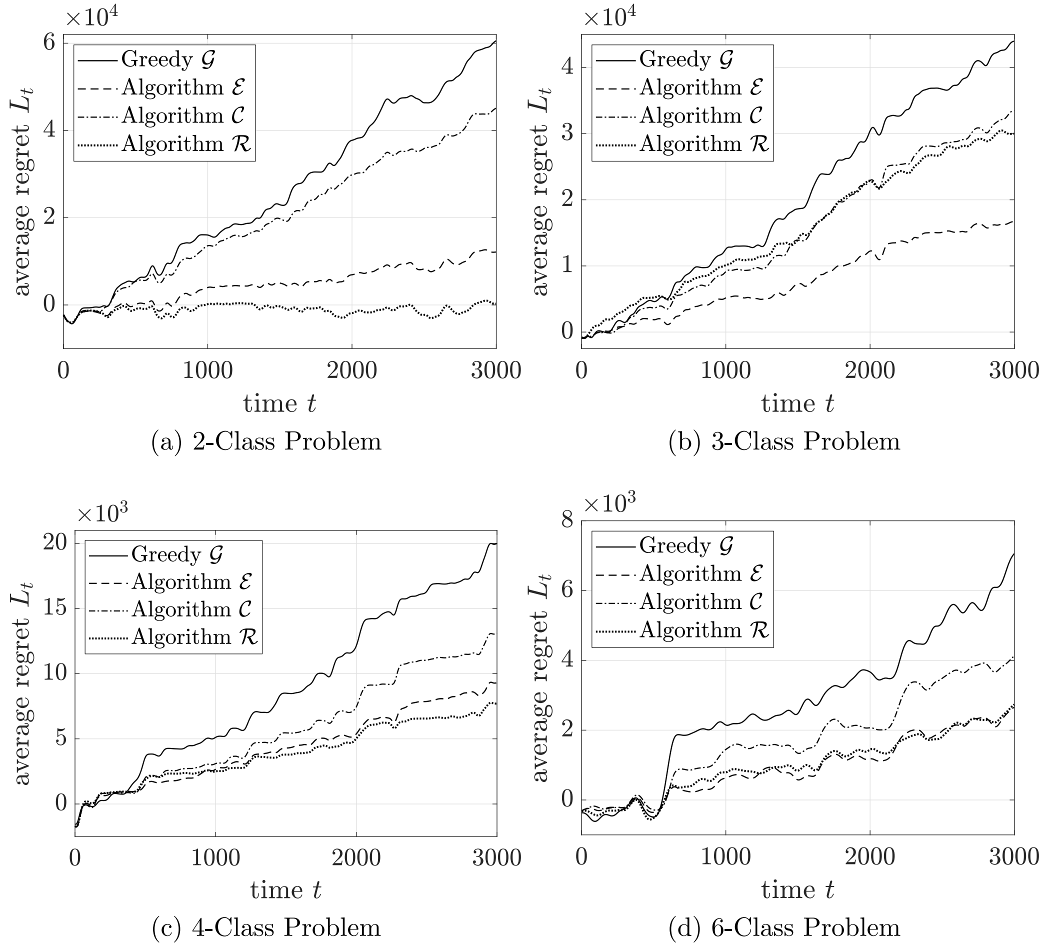
Figure 6.3: Effect of the Postprocessing step in the Online Framework

Figure 6.3 studies this in more detail by analysing the benefit caused by the postprocessing step. It shows clearly that the postprocessing step does in fact significantly improve the optimization result in the online context. It is therefore strongly advised to be incorporated and in the following the offline algorithm $\mathcal{O}$ and its variation $\mathcal{O}'$ will always include this exact postprocessing step. Figure 6.3a also supports the assumption that the optimality of the local optimization algorithm on the 2-class problem is a result of the postprocessing step.

6.3 Stochastic Optimization Algorithms

Now a closer look is taken at the algorithms incorporating online stochastic combinatorial optimization, defined in Chapter 2.2. When recalling the

general definition of the algorithmic framework of stochastic optimization algorithms, some parameters have been defined. These parameters include the sample horizon h_s and the number of optimization steps n. Recall that the sample horizon h_s defines how many future request sets are sampled from the available distribution and n defines the number of overall executions of algorithm $\mathcal{O}'$ at each time step.



(a) 2-Class Problem

(b) 3-Class Problem

(c) 4-Class Problem

(d) 6-Class Problem

Figure 6.4: Comparison of the Stochastic Optimization Algorithms

The effect of changing both parameters has been extensively studied in [1] and is therefore not repeated. As expected, increasing either parameter improves the algorithms but according to the study there is not much benefit setting the parameters $n > 150$ and $h_S > 100$. In this work the parameters are set to $n = 100$ and $h_s = 75$. The stochastic optimization algorithms incorporate sampling the underlying input distribution $\mathcal{I}$. Figure 6.4 shows

a comparison of the stochastic optimization algorithms under the assumption that the underlying distribution is fully observable. This is a very strong assumption but for only comparing the algorithmic behaviour it is a reasonable choice. As expected, every algorithm can outperform greedy and the consensus algorithm $\mathcal{C}$ has the worst performance in any setting. The expectation algorithm $\mathcal{E}$ has a significantly better performance in any setting compared to the consensus algorithm. Interesting is the behaviour of the regret algorithm $\mathcal{R}$. Its performance seems to be strongly dependent on the problem. On the 2-class and 4-class problem algorithm $\mathcal{R}$ performs better than algorithm $\mathcal{E}$, on the 3-class problem and 6-class problem the regret algorithm cannot outperform algorithm $\mathcal{E}$. The analysis made in [1] shows that algorithm $\mathcal{R}$ should always outperform the expectation algorithm, however note that now in this work the packet weights are drawn randomly (in [1] they have been chosen arbitrarily with considerable weight differences) and the Markov models describing the packet arrival are not normalised (in [1] the models have been normalised so that the expected number of packets per time step is 1.5). Clearly, algorithm $\mathcal{R}$ can achieve significant improvements, e.g. for the 2-class problem it is almost optimal, but it is necessary to emphasize that its performance is less robust than that of other algorithms.

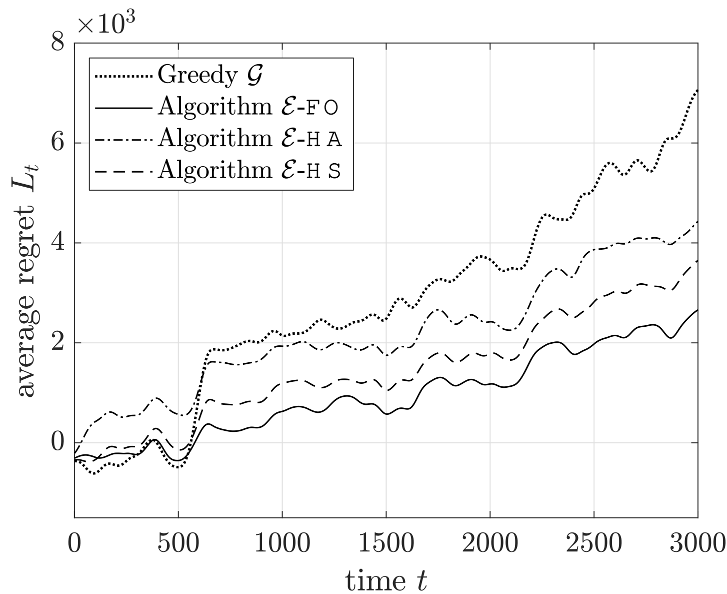


Figure 6.5: Sampling Methods for the Stochastic Optimization Algorithms

In the scope of this work there have been presented some techniques to learn the underlying distribution $\mathcal{I}$ or to cope with the fact that no knowledge at all on the packet arrival can be assumed. Figure 6.5 shows the effect of the

sampling method on the performance on the 6-class problem. It compares the case of full observation (FO) with the two methods that assume no knowledge on the input distribution: historical averaging (HA) and historical sampling (HS). The learning horizon was set to $\lambda = 500$ for both methods. Section 5.1 also discussed a way of learning the underlying distribution $\mathcal{I}$ for the case that it can be assumed to be composed of Markov models (Figure 5.2). This approach can perform arbitrarily close to the method $\mathcal{E}$-FO, but note that this knowledge is also a very strong assumption. In this work the packet arrival is in fact modelled using Markov models, however for the analysis it is more interesting to not assume that this is known, especially when comparing the stochastic optimization algorithms with the reinforcement learning algorithms that do not make this assumption. As expected, the historical sampling method is significantly better than simply taking an average over the past λ arrivals. In this way the structural sequences in the data are preserved and incorporated in the sampling procedure.

6.4 Reinforcement Learning Algorithms

For the reinforcement learning algorithms a key performance factor is efficient exploration. For this, first the ϵ-greedy and the UCB-1 algorithms are compared in Figure 6.6.

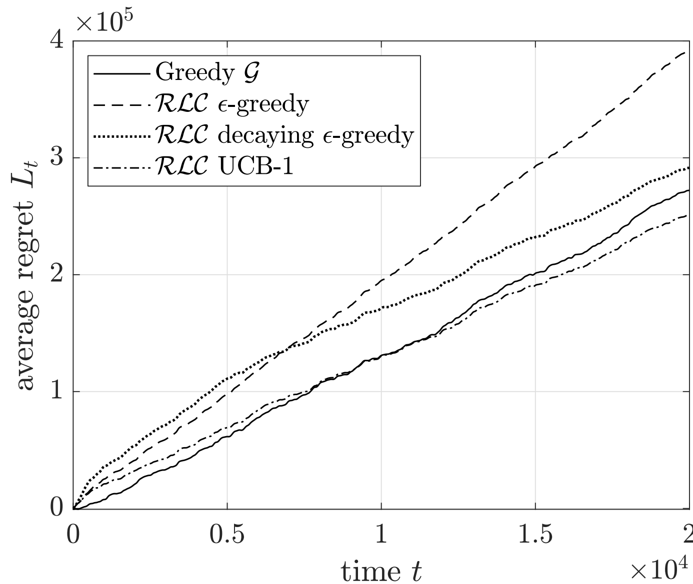
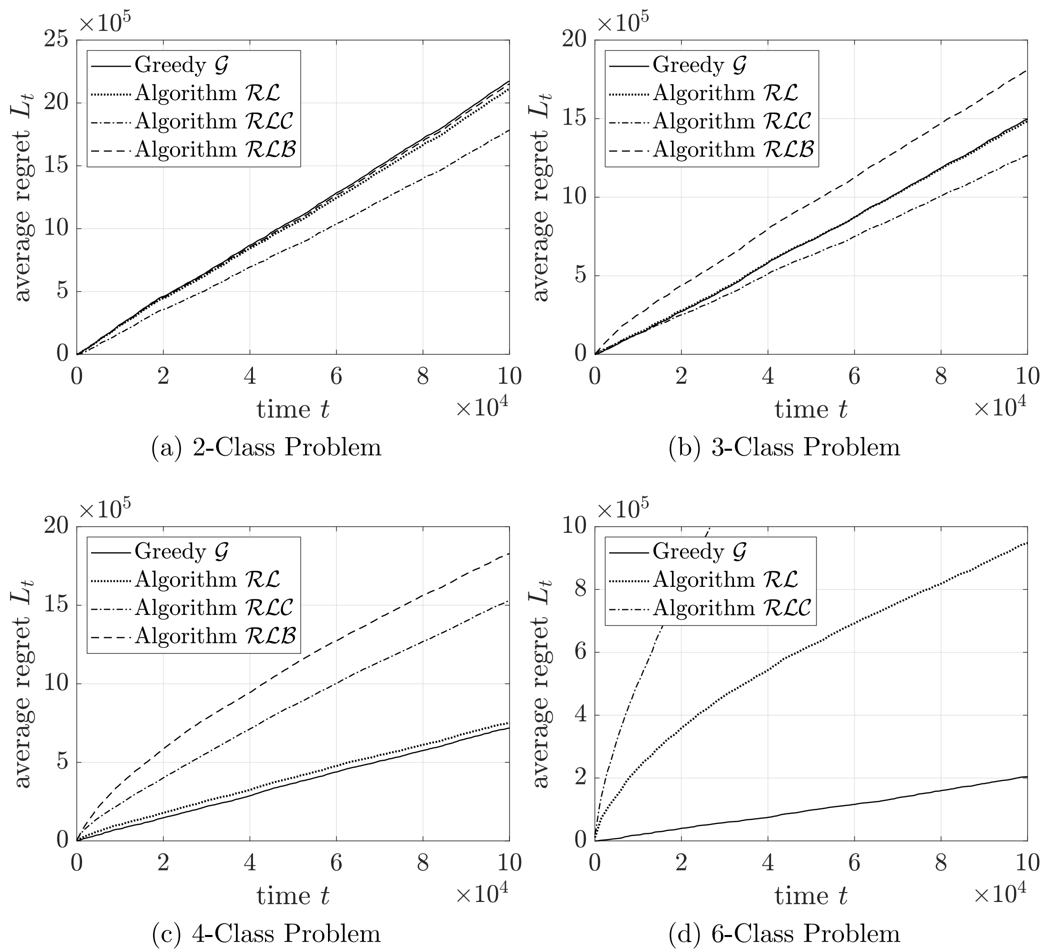


Figure 6.6: Initial Exploration Loss on the 3-Class Problem

For the ϵ-greedy algorithm also a strategy with decaying ϵ is considered in

order to decrease exploration (and therefore also packet loss) as time passes. This is reasonable since after an initial learning phase, the algorithm is expected to have achieved a reasonable approximation of the optimal action value function. From this point, exploration is advised to be reduced. For the comparison, the conservative algorithm $\mathcal{RLC}$ is applied on instances of the 3-class problem. The exploration parameters have been set to $c = 2$ for the UCB-1 algorithm and $\epsilon = 0.05$ for the ϵ-greedy strategy. The decaying ϵ-greedy algorithm starts with $\epsilon = 0.25$ and gradually reduces the exploration parameter to $\epsilon = 0$ within 5000 time steps.



(a) 2-Class Problem

(b) 3-Class Problem

(c) 4-Class Problem

(d) 6-Class Problem

Figure 6.7: Comparison of the Reinforcement Learning Algorithms

The comparison shows that the classical ϵ-greedy strategy is not suitable. Also, for the decaying ϵ-greedy strategy the initial accumulated loss is large. It eventually achieves a similar approximation of the action value function

as the UCB-1 algorithm, but it cannot recover the initially accumulated loss. UCB-1 obviously incorporates some exploration, but it converges very fast. This shows that a directed exploration can perform significantly better than random exploration.

In the following now the different variants of the proposed reinforcement learning algorithms are compared. They all incorporate the UCB-1 strategy with the exploration parameter set to $c = 2$. The analysis in Figure 6.7 shows that for the n-class problem with $n < 4$, the conservative Q-learning algorithm $\mathcal{RLC}$ is the best approach. It significantly reduces the accumulated regret over time. In any setting, the algorithm $\mathcal{RLB}$ that incorporates belief states performs worst. Note that for the conducted experiments it was assumed that the input distribution is fully observable. If in this case the algorithm $\mathcal{RLB}$ does not achieve reasonable results, learning the belief states over time will be even worse. It is conjectured that in this representation the state space is too large and is therefore not able to converge within a reasonable time horizon. The classical Q-learning algorithm performs very similar to the greedy algorithm. In Figure 6.7b it is almost not visible, but at the end it achieved a little improvement compared to greedy. However, for the n-class problem with $n \geq 4$, none of the proposed algorithms can outperform the greedy algorithm. A possible reason might be that the state space gets too large for more complex problems including more distinct packet classes.

Chapter 7

Conclusion

In this work several algorithms have been proposed to incorporate anticipative behaviour for scheduling problems in the online framework. The algorithms have been experimentally studied on a packet scheduling problem with packet arrivals modelled by Markov models. The analysis showed that algorithms using online stochastic combinatorial optimization yield the smallest cumulative weighted packet loss over time in any setting. The regret algorithm $\mathcal{R}$ can have the significantly best performance, however it is also less robust compared to the expectation algorithm $\mathcal{E}$ and potentially sensitive to specific problems or weight distributions. The consensus algorithm $\mathcal{C}$ turned out to be the least suitable algorithm for the packet scheduling problem studied in this work. When no knowledge on the underlying distribution is assumed, the method of historical sampling showed to be the best approach to generate possible future requests.

The proposed reinforcement learning algorithms also achieved to outperform the greedy algorithm, however only slightly. The algorithm $\mathcal{RLB}$, that incorporates belief states in its state representation, turned out to be not suitable and ineffective for the problem. When only a few distinct packet classes are arriving, algorithm $\mathcal{RLC}$ is the best approach. However, when more packet classes are present, none of the proposed reinforcement learning algorithms converges within a reasonable time horizon and the classical Q-learning algorithm $\mathcal{RL}$ showed to be the best approach.

When comparing the approaches discussed in this work, two main properties must be mentioned. First, the performance of stochastic optimization is significantly better compared to reinforcement learning algorithms in any problem setting. When using the stochastic optimization approach, the cumulative weighted packet loss can be strongly reduced over time. However, stochastic optimization requires also a significantly larger runtime to make a decision at each time step since it requires to execute the offline optimization

algorithm multiple times at each time step. Therefore, if low computing power or severe time restrictions must be expected, reinforcement learning might be better suitable. However, it must be mentioned that the comparison of the two approaches is not completely fair. Because of the sampling at each time step the stochastic optimization algorithms have much more information available. To perform a conclusive comparison, the reinforcement learning algorithm should also have all these sampled requests available for learning. This would drastically reduce the time until the Q-learning algorithm converges.

In the scope of this work the packet arrival has been only modelled using Markov models. It might be also interesting to study the proposed algorithms on instances generated using real network traffic data. It is open to study if also in that case the sampling procedure, required when applying stochastic optimization, is effective. Also, using machine learning to generate possible future requests has not been further discussed. This could further improve the performance of the stochastic optimization algorithms while still making no assumptions on the model of the underlying distribution.

Also, only one possible state representation for the reinforcement learning algorithm has been studied, which eventually showed not to be suitable for more complex instances of the packet scheduling problem. Experimenting with different representations could further improve the performance and eventually make the reinforcement learning algorithms also applicable to problems containing a larger class quantity.

Bibliography

[1] Pascal van Hentenryck and Russell Bent. *Online stochastic combinatorial optimization*. MIT Press, Cambridge, Mass, 2006.

[2] Susanne Albers. Online algorithms. In Dina Goldin, Scott A. Smolka, and Peter Wegner, editors, *Interactive Computation: The New Paradigm*, pages 143–164. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[3] Stephan Meisel. *Anticipatory optimization for dynamic decision making*, volume 51 of *Operations research/computer science interface series*. Springer, New York, NY, 2011.

[4] Luc Mercier and Pascal van Hentenryck. Performance analysis of online anticipatory algorithms for large multistage stochastic integer programs. In *IJCAI*, pages 1979–1984, 2007.

[5] Robert Rosen. *Anticipatory systems: Philosophical, mathematical and methodological foundations*, volume 1 of *IFSR international series on systems science and engineering*. Pergamon Press, Oxford u.a., 1985.

[6] Martin V. Butz, Olivier Sigaud, and Pierre Gérard. *Anticipatory Behavior in Adaptive Learning Systems: Foundations, Theories, and Systems*, volume 2684 of *Lecture Notes in Computer Science*. Springer, Berlin and Heidelberg, 2003.

[7] M. A. L. Thathachar and P. S. Sastry. *Networks of Learning Automata: Techniques for Online Stochastic Optimization*. Springer US, Boston, MA, 2004.

[8] Haruhiko Suwa and Hiroaki Sandoh. *Online scheduling in manufacturing: A cumulative delay approach*. Springer, London, 2013.

[9] Hyeong Soo Chang, Robert Givan, and Edwin K. P. Chong. Online scheduling via sampling. In *Proceedings of the Fifth International*

Conference on Artificial Intelligence Planning Systems, AIPS'00, pages 62–71. AAAI Press, 2000.

[10] Russell Bent and Pascal van Hentenryck. Online stochastic and robust optimization. In *Annual Asian Computing Science Conference*, pages 286–300, 2004.

[11] Martin Böhm, Marek Chrobak, Łukasz Jeż, Fei Li, Jiří Sgall, and Pavel Veselý. Online packet scheduling with bounded delay and lookahead. *Theoretical Computer Science*, 776:95–113, 2019.

[12] Michael Markovitch and Gabriel Scalosub. Bounded delay scheduling with packet dependencies. In *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 257–262, 2014.

[13] Susanne Albers. Online algorithms: a survey. *Mathematical Programming*, 97(1):3–26, 2003.

[14] Amos Fiat and Gerhard J. Woeginger. *Online Algorithms: The State of the Art*, volume 1442 of *Lecture Notes in Computer Science*. Springer, Berlin and Heidelberg, 1998.

[15] Elias Koutsoupias and Christos H. Papadimitriou. Beyond competitive analysis. *SIAM Journal on Computing*, 30(1):300–317, 2000.

[16] S. Ben-David. *On the power of randomization in online algorithms*, volume 90,23 of *TR / International Computer Science Institute*. International Computer Science Inst, Berkeley, Calif., 1990.

[17] Lachlan Andrew, Siddharth Barman, Katrina Ligett, Minghong Lin, Adam Meyerson, Alan Roytman, and Adam Wierman. A tale of two metrics: Simultaneous bounds on competitiveness and regret. In *Conference on Learning Theory*, pages 741–763, 2013.

[18] Amit Daniely and Yishay Mansour. Competitive ratio vs regret minimization: achieving the best of both worlds. In Aurélien Garivier and Satyen Kale, editors, *Proceedings of the 30th International Conference on Algorithmic Learning Theory*, volume 98 of *Proceedings of Machine Learning Research*, pages 333–368, Chicago, Illinois, 2019. PMLR.

[19] Frank Werner, Larysa Burtseva, and Yuri Sotskov, editors. *Algorithms for scheduling problems*. MDPI, Basel and Beijing and Wuhan and Barcelona and Belgrade, 2018.

[20] Kurt Marti. *Stochastic optimization methods: Applications in engineering and operations research*. Springer Berlin Heidelberg, Berlin, Heidelberg and s.l., 3rd ed. 2015 edition, 2015.

[21] Russell Bent and Pascal van Hentenryck. Online stochastic optimization without distributions. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling*, volume 5, pages 171–180. AAAI Press, 2005.

[22] John R. Birge and François Louveaux. *Introduction to stochastic programming*. Springer Series in Operations Research and Financial Engineering. Springer, New York, NY, 2. ed. edition, 2011.

[23] Georg Ch Pflug and Alois Pichler. *Multistage Stochastic Optimization*. Springer Series in Operations Research and Financial Engineering. Springer International Publishing, Cham and s.l., 2014.

[24] Peter A. N. Bosman. Learning, anticipation and time-deception in evolutionary online dynamic optimization. In *Proceedings of the 7th annual workshop on Genetic and evolutionary computation*, pages 39–47, 2005.

[25] Peter A. N. Bosman and Han La Poutre. Learning and anticipation in online dynamic optimization with evolutionary algorithms: the stochastic case. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1165–1172, 2007.

[26] Csaba Szepesvári. *Algorithms for Reinforcement Learning*, volume #9 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool, San Rafael, 2010.

[27] Marco Wiering and Martijn van Otterlo. *Reinforcement Learning: State-of-the-Art*, volume 12 of *Adaptation, Learning, and Optimization*. Springer, Heidelberg, 2012.

[28] Bharathan Balaji, Jordan Bell-Masterson, Enes Bilgin, Andreas Damianou, Pablo Moreno Garcia, Arpit Jain, Runfei Luo, Alvaro Maggiar, Balakrishnan Narayanaswamy, and Chun Ye. Orl: Reinforcement learning benchmarks for online stochastic optimization problems. *arXiv preprint arXiv:1911.10641*, 2019.

[29] Herman L. Ferrá, Ken Lau, Christopher Leckie, and Anderson Tang. Applying reinforcement learning to packet scheduling in routers. In

Proceedings of the 15th Innovative Applications of Artificial Intelligence Conference, pages 79–84. AAAI Press, 2003.

[30] Richard S. Sutton and Andrew Barto. *Reinforcement learning: An introduction.* Adaptive computation and machine learning. The MIT Press, Cambridge, MA and London, second edition edition, 2018.

[31] Stuart Russell. Learning agents for uncertain environments. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 101–103, 1998.

[32] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards: Ph.D.Thesis.* King's College, Cambridge, 1989.

[33] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[34] Arryon D. Tijsma, Madalina M. Drugan, and Marco A. Wiering. Comparing exploration strategies for q-learning in random stochastic mazes. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2016.

[35] Richard Dearden, Nir Friedman, and Stuart Russell. Bayesian q-learning. In *Proceedings of the 15th National Conference on Artificial Intelligence*, pages 761–768. AAAI Press, 1998.

[36] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[37] Thomas Jaksch, Ronald Ortner, and Peter Auer. Near-optimal regret bounds for reinforcement learning. *Journal of Machine Learning Research*, 11(4), 2010.

[38] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[39] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative Q-learning for offline reinforcement learning. *arXiv preprint arXiv:2006.04779*, 2020.

[40] Jan Karel Lenstra, A. RinnooyH.G. Kan, and Peter Brucker. Complexity of machine scheduling problems. In *Annals of discrete mathematics*, volume 1, pages 343–362. Elsevier, 1977.

[41] Nikhil Bansal, Avrim Blum, Shuchi Chawla, and Adam Meyerson. Online oblivious routing. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 44–49, 2003.

[42] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 2014.

[43] Elham Semsar-Kazerooni and Khashayar Khorasani. *Team Cooperation in a Network of Multi-Vehicle Unmanned Systems: Synthesis of Consensus Algorithms*. Springer, New York, NY, 2013.

[44] Russell Bent and Pascal van Hentenryck. The value of consensus in online stochastic scheduling. In *Proceedings of the 14th International Conference on Automated Planning*, volume 4, pages 219–226. AAAI Press, 2004.

[45] Russell Bent, Irit Katriel, and Pascal van Hentenryck. Sub-optimality approximations. In *International Conference on Principles and Practice of Constraint Programming*, pages 122–136, 2005.

[46] Marlos C. Machado, Sriram Srinivasan, and Michael Bowling. Domain-independent optimistic initialization for reinforcement learning. *arXiv preprint arXiv:1410.4604*, 2014.

[47] Danilo Bzdok, Naomi Altman, and Martin Krzywinski. Statistics versus machine learning. *Nature Methods*, 15, 2018.

[48] Olivier Cappé, Eric Moulines, and Tobias Rydén. *Inference in hidden Markov models*. Springer series in statistics. Springer Science & Business Media, 2005.

[49] Brigham Anderson and Andrew Moore. Active learning for hidden markov models: Objective functions and algorithms. In *Proceedings of the 22nd international conference on Machine learning*, pages 9–16, 2005.

[50] Leonard E. Baum et al. An inequality and associated maximization technique in statistical estimation for probabilistic functions of markov processes. *Inequalities*, 3(1):1–8, 1972.

[51] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[52] Achim G. Hoffmann et al. General limitations on machine learning. In *ECAI*, pages 345–347, 1990.

[53] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass, second edition edition, 2018.

[54] Aileen Nielsen. *Practical time series analysis: Prediction with statistics and machine learning*. O'Reilly Media, Inc., Sebastopol, CA, first edition edition, 2019.