Scheduling and Packing Under Uncertainty

vorgelegt von

Franziska Eberle, M.Sc.

geboren in Kempten (Allgäu)

Vom Fachbereich 3 – Mathematik und Informatik
der Universität Bremen
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
– Dr. rer. nat. –

genehmigte Dissertation

Gutachter: Prof. Dr. Nicole Megow
Prof. Dr. Anupam Gupta

Tag der wissenschaftlichen Aussprache: 13. November 2020

Bremen 2020

Acknowledgements

I am grateful beyond words for the support of my family, friends, and colleagues which made this thesis possible in the first place. First and foremost, I thank my advisor Nicole for her constant encouragement, support, and the always open office door. Telling me about optimization under uncertainty, inviting me to join her in Bremen, and introducing me to this research community are only some parts that contributed to the very inspiring environment.

Many thanks also go to Cliff for welcoming me in New York for a research visit, which led to co-authoring one paper. Further, I am very grateful to Anupam for taking the second assessment of this thesis.

Moreover, I thank my colleagues, most of whom I now call friends, for the great collaboration and for reading parts of this thesis. I especially appreciated being welcomed with open arms at my very first MAPSP more than three years ago and the great atmosphere that our early (sorry!) morning runs and late evening discussions created at every single conference or workshop. I owe special thanks to Lukas, for completely reading this thesis and for the conversations about all topics, including research and dinner plans.

I am also very grateful for my friends and the many hours spent laughing, cooking, hiking, skiing, playing games, bouldering, going for a run, exploring new and old cities, and discussing crazy ideas about life. Special thanks go to Lena and Isa for proofreading parts of this thesis and for always believing in my abilities.

Moreover, I thank my family for their constant support and for never questioning my path, even if it led me and will lead me far away from home. In particular, I thank my parents for teaching me to never stop asking questions or seeking answers. Last but not least, I thank my little brother for proofreading parts of this thesis and, in general, for always having my back and being up for new adventures.

Flo, although the last months have been incredibly hard for you, you still pushed me to finish my thesis. I would not be at this point without you. This is for you!

Bremen, December 2020 Franziska Eberle

Table of Contents

1 Introduction

Incomplete information is a major challenge when translating combinatorial optimization results to recommendations for real-world applications since problem relevant parameters change frequently or are not known in advance. A particular solution may perform well on some specific input data or estimation thereof, but once the data is slightly perturbed or new tasks need to be performed, the solution may become arbitrarily bad or even infeasible. Thus, either solving the problem under uncertainty or efficiently updating the solution becomes a necessity. This thesis explores several models for uncertainty in various problems from two fundamental fields of combinatorial optimization: scheduling and packing.

Scheduling problems cover a variety of real-world applications. They arise whenever scarce resources have to complete a set of tasks while optimizing some objective. Possible applications range from the industrial sector with production planning via the service sector with delivery tasks to the information sector with data processing and cloud computing. Given the undeniable effects of climate change we are facing today and the rising pressure to cut costs to keep up with competitors, efficient solutions to any of these problems are paramount. Efficiency might refer to the minimal necessary duration for which the system is running to process all tasks or the maximal achievable throughput in a given time interval.

Packing problems typically appear whenever items have to be assigned to resources with capacities. The most obvious applications are transportation processes, e.g., loading of trucks, that take place at every stage of the manufacturing process until the delivery to the client. Further, packing problems can also be found in computing clusters and financial applications: Capacities have to be obeyed and costs have to be minimized, when assigning virtual machines to real servers in computing clusters. Or, when making new investment decisions, certain aspects, such as risk or overall volume, have to be bounded while maximizing profit. These examples can all be considered as packing problems since they require compliance with certain capacity constraints while maximizing the overall value of successfully packed items.

Incomplete information may be caused by various reasons, such as the unpredictable arrival of new tasks or having only estimates of input parameters at hand. In any of these cases, we are interested in finding provably good solutions in reasonable time. In other words, we would

like to design *algorithms* that deal with incomplete information while performing sufficiently well. This thesis focuses on three models of uncertainty in the input.

(i) If the precise input is unknown and only estimates of the relevant parameters are given, the setting can be modeled via random variables, whose specific outcome is unknown but which provide some knowledge about possible scenarios. While there are various branches of stochastic optimization, we focus on *stochastic* scheduling.

(ii) Models where the data of an instance is only incrementally revealed and no full knowledge about the instance is given in advance are called *online*. Since waiting for the last piece of information to be revealed is often impossible or infeasible, immediate and possibly irrevocable actions have to be taken as new information is revealed.

(iii) Lastly, problems where the input is subject to small perturbations such as the deletion or addition of objects from or to an instance are called *dynamic*. Since only minor changes happen, we are interested in quickly computing a new solution of good quality based on the previously obtained results.

Outline of the Thesis

We investigate several combinatorial optimization problems and develop algorithms that perform provably well under incomplete information. Chapter 2 gives a brief overview over the necessary concepts for analyzing algorithms. Moreover, we introduce the problem types considered and how to formally model uncertainty. We mostly measure the performance of an algorithm in terms of the quality of the solution it finds. To this end, we also briefly explain how to adapt the analysis of algorithms when facing uncertainty in the input. The remainder of the thesis is organized as follows.

Stochastic Scheduling

In Chapter 3, we investigate a scheduling problem where the job set is given in advance but (almost) no information about the duration, the *processing time*, of a job is available. More precisely, the processing times are modeled as independent random variables and the scheduler only has access to their distributional information but not to their exact realization. The goal is to schedule the jobs on m parallel machines such that the expected value of the total completion time is minimized. We rule out distribution-independent performance guarantees for a highly useful and widely used class of algorithms, so-called *index policies*. We also show that giving the algorithm slightly more freedom in choosing the scheduling order of the jobs enables us to obtain a performance guarantee linear in m for the type of instances we used in our negative result. We complement this with a closer look at these special instances and providing a set of rules for which even constant performance guarantees are possible.

Online Scheduling with Reassignment

In Chapter 4, we consider an online scheduling problem where n jobs are incrementally revealed and, on arrival, have to be assigned immediately to one of the machines to minimize the latest completion time of a job. However, each job may only be processed by a subset of the machines. There exist strong lower bounds on the performance of any online algorithm, that has to irrevocably assign jobs on arrival, by Azar, Naor, and Rom [ANR92]. Therefore, Gupta, Kumar, and Stein [GKS14] weaken the traditional irrevocability assumption for online algorithms and allow their algorithm to moderately reassign jobs in order to maintain a good solution. They bound the number of such changes which is usually referred to as online optimization with *recourse*. We are able to generalize their result to the setting where each job comes with its individual *cost* that has to be paid upon (re)assigning it. We obtain a matching performance guarantee linear in $\log\log(mn)$ with a bound on the incurred reassignment cost linear in the total assignment cost. This model also generalizes online optimization with *migration* that is concerned with bounding the volume of changes an online algorithm makes.

Online Deadline-Sensitive Scheduling

In Chapters 5 and 6, we investigate a scheduling problem where jobs arrive online over time at their *release date* and have to be processed by identical parallel machines before their respective *deadlines*. The goal is to maximize the *throughput*, i.e., the number of jobs that complete before their deadlines. As shown by Baruah, Haritsa, and Sharma [BHS94], hard instances for online algorithms involve "tight" jobs, i.e., jobs that need to be scheduled immediately and without interruption in order to complete on time. To circumvent these difficulties, we require that the instance does not contain tight jobs and thus enforce some relative slack for each job in the interval defined by its release date and its deadline. We assume that each job's interval has length at least $(1+\varepsilon)$ times its processing time for a given slackness parameter $\varepsilon > 0$.

In Chapter 5, we develop an online algorithm for maximizing the throughput. In fact, this algorithm is quite similar to the one designed by Lucier et al. [LMNY13] for maximizing the total weight of jobs finished on time. With a completely different analysis, we show that its worst-case performance depends linearly on $\frac{1}{\varepsilon}$ in the unweighted case. By giving a matching lower bound for *any* online algorithm, we also prove that this algorithm is best possible.

In Chapter 6, we introduce the notion of *commitment* to the model. That is, any scheduler either has to commit to the completion of a job at some point between its release date and its deadline or discard the job completely. We investigate the impact various commitment requirements have on the performance of online algorithms and we rule out any online algorithm with reasonable performance if the commitment is required immediately upon arrival of a job. For two less strict commitment requirements, we develop an algorithm with provably good performance. Surprisingly, when the scheduler has to commit upon starting a job, we obtain the same asymptotic performance guarantee as in the setting without commitment. Requir-

ing δ-commitment means that the commitment decision has to be made when the slackness assumption reduces from ε to δ, for $\delta \in (0, \varepsilon)$. Since this commitment requirement tightens to commitment upon arrival when δ tends to ε, it is not surprising that the performance guarantee diverges with increasing δ. However, if δ is bounded away from ε, then we recover the same performance guarantee as in the model without commitment. We supplement this chapter with further lower bounds for online scheduling with commitment.

Dynamic Packing

In Chapter 7, we design and analyze a dynamic algorithm for knapsack problems. Here we are given a set of items with sizes and values as well as a set of knapsacks with capacities. Both sets are subject to small changes, specifically, the deletion or addition of an item or a knapsack. The goal is to maintain an assignment of items to knapsacks that does not exceed their capacities and maximizes the total value of packed items. We give a dynamic algorithm that deals with a change in the instance in poly-logarithmic time while maintaining an almost optimal solution. At the heart of our result lies a novel and dynamic approach to linear grouping of items. For the special case of many identical knapsacks, we can do even better and give a significantly faster algorithm. We also show that it is impossible to obtain similarly fast algorithms for few knapsacks, unless $\mathcal{P} = \mathcal{NP}$.

2

Preliminaries

We introduce notation and concepts used throughout this thesis. Further, we give a short overview over *scheduling* and *packing* problems as well as a brief introduction to the concepts of uncertainty considered.
We assume some knowledge of the basic concepts of combinatorial optimization and refer to the books by Korte and Vygen [KV02] and by Schrijver [Sch03] for an overview. For an in-depth introduction to network flows, we point to the book by Ahuja, Magnanti, and Orlin [AMO93]. For a introduction to the theory of linear programming, we recommend the textbook by Bertsimas and Tsitsiklis [BT97]. Further, we assume some familiarity with probability theory and refer to the books by Biagini and Campanino [BC16] and Gut [Gut13]. For an overview over discrete probability distributions, we recommend the book by Johnson, Kemp, and Kotz [JKK05] and the references therein.

Table of Contents

2.1 Basic Notation

We use $\mathbb{N}_0$ and $\mathbb{N}$ to refer to the set of natural numbers with and without 0, respectively. Further, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ denote the sets of integral, rational, and real numbers, respectively. By $\mathbb{Z}^+$, $\mathbb{Q}^+$ and $\mathbb{R}^+$ we refer to the sets of non-negative integral, rational, and real numbers,

respectively. For a natural number $n \in \mathbb{N}$, we define $[n]$ to be the set of all natural numbers up to n, that is, $[n] = \{1, \dots, n\}$. For two real numbers x and $y \in \mathbb{R}$, let $[x, y]$ be the closed interval, (x, y) the open interval, and $[x, y)$ as well as $(x, y]$ the half-open intervals from x to y in $\mathbb{R}$; note that any of these intervals is empty if $y < x$ and only $[x, y]$ is non-empty if $x = y$. We use $\log x$ to refer to $\log_2 x$ for some $x > 0$.

Let $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}^+$. We use the Big-$\mathcal{O}$ Notation for classifying the asymptotic growth of a function f in terms of a function g when x tends towards infinity. Formally, if there is a $x_0 \geq 0$ and a constant $c \geq 0$ such that $f(x) \leq cg(x)$ for all $x \geq x_0$, then $f \in \mathcal{O}(g)$. Similarly, if there is a $x_0 \geq 0$ and a constant c with $f(x) \geq cg(x)$ for all $x \geq x_0$, then $f \in \Omega(g)$. If $f \in \mathcal{O}(g) \cap \Omega(g)$, then $f \in \Theta(g)$. Intuitively, if f does not grow faster than g does, then $f \in \mathcal{O}(g)$, while f not growing *slower* than g means $f \in \Omega(g)$. If these two functions asymptotically grow with the same speed, then $f \in \Theta(g)$.

2.2 Algorithm Analysis and Complexity

In this thesis, we design and analyze algorithms for scheduling and packing problems, which we informally introduced in the previous chapter and which will be formally defined in the next section. In general, an *algorithm* is a finite sequence of instructions that solves a particular problem. It takes an instance of a given problem as input and returns a solution for this instance after finitely many computational steps. The performance of an algorithm can be measured, e.g., in terms of its time complexity and the quality of its solution. For evaluating the time complexity of an algorithm, we compare its *running time*, i.e., the number of basic computational steps, with the size of the input data under a particular encoding scheme; we usually assume binary encoding of the data. Ideally, we design an algorithm that obtains a provably optimal solution with worst-case running time that is bounded by a polynomial in the input size. If a problem admits such a *polynomial-time* algorithm that finds the optimum for all instances, then its *decision variant* belongs to the class $\mathcal{P}$ of *polynomial-time solvable* problems. The decision variant of an optimization problem poses the question of whether the optimal solution value is at least or at most a certain value $f^* \in \mathbb{Z}$. Note that rational numbers can be ignored due to scaling, and we omit irrational numbers since encoding them in bounded time is impossible.

For most combinatorial problems investigated in this thesis, it is not known whether they admit optimal polynomial-time algorithms. In fact, it is widely believed that these problems are not polynomial-time solvable. For clarifying the notion of computational complexity, let us focus on decision problems, that is, problems for which the answer is either YES or NO. If there is a certificate, e.g., a solution, such that one can *verify* that the correct answer is YES in polynomial time based on this certificate if and only if the instance is a YES-instance, then this problem belongs to the class $\mathcal{NP}$ of non-deterministically polynomial-time solvable problems. Intuitively, non-deterministic implies that, if we are able to guess a correct solution and the

instance is a YES-instance, then we can check this in polynomial time. Clearly, all problems in $\mathcal{P}$ are also in $\mathcal{NP}$ since we can use the polynomial-time algorithm corresponding to the problem to solve and thus decide the problem. Ever since Cook [Coo71] and Karp [Kar72] laid the foundations of complexity theory, it is a major open question whether all problems in $\mathcal{NP}$ admit polynomial-time algorithms or whether $\mathcal{P}$ is a proper subset of $\mathcal{NP}$. A particularly interesting subset of problems in $\mathcal{NP}$ is the set of $\mathcal{NP}$-*complete* problems since they act as representatives of the entire class $\mathcal{NP}$. If one $\mathcal{NP}$-complete problem admits a polynomial-time algorithm, then *all* problems in $\mathcal{NP}$ are polynomial-time solvable and $\mathcal{P} = \mathcal{NP}$.

An *optimization problem* describes the task to optimize a certain objective function subject to some constraints. We observe that the decision and the optimization variants are equivalent in the following sense: If we have an algorithm for one of them, then we can translate it to an algorithm for the other in polynomial time as follows. On the one hand, having an algorithm finding an optimal solution, we simply compare the returned solution value to f^*, the parameter of the decision problem. On the other hand, having an algorithm for the decision variant, we can use binary search over f^* to solve the optimization problem. This leads to the notion of $\mathcal{NP}$-*hardness* for all problems that are at least as hard to solve as any $\mathcal{NP}$-complete problem, i.e., a polynomial-time algorithm for a $\mathcal{NP}$-hard problem implies $\mathcal{P} = \mathcal{NP}$. In particular, the optimization variants of $\mathcal{NP}$-complete problems are $\mathcal{NP}$-hard.

A particular class of $\mathcal{NP}$-hard problems are the so-called *strongly* $\mathcal{NP}$-hard problems: They remain $\mathcal{NP}$-hard even if the appearing numbers are polynomially bounded in the input size. (A number is bounded exponentially in the length of its binary encoding.) In particular, unless $\mathcal{P} = \mathcal{NP}$, such problems do not admit *pseudopolynomial-time* algorithms, i.e., algorithms whose running time is polynomially bounded in the size of the input and the appearing numbers in the input. For a thorough introduction to complexity theory, please refer to the book by Garey and Johnson [GJ79].

Approximation algorithms Since it is widely assumed that $\mathcal{P} \neq \mathcal{NP}$, we cannot hope to find an optimal polynomial-time algorithm for every instance of a $\mathcal{NP}$-hard problem. In this thesis, we relax the notion of optimality and develop polynomial-time algorithms that find provably "good" solutions. More precisely, for a minimization (maximization) problem, we are interested in a polynomial-time algorithm for which we can prove that, for every instance, the objective value of the returned solution is at most α $\left(\text{at least } \frac{1}{\alpha}\right)$ times the value of an optimal solution for some $\alpha \geq 1$. In this case, we say that the algorithm is an α-*approximation algorithm* or short α-*approximation*. The infimum α such that the algorithm is an α-approximation is called the *approximation ratio* or *approximation factor* of the algorithm.

An approximation scheme is a family of polynomial-time algorithms $(\mathcal{A}_\varepsilon)_{\varepsilon>0}$ such that, for every $\varepsilon > 0$, $\mathcal{A}_\varepsilon$ is a $(1+\varepsilon)$-approximation algorithm. Based on how well the running time scales with decreasing parameter ε, we distinguish *Polynomial Time Approximation Schemes (PTAS)* with arbitrary dependency on ε, *Efficient Polynomial Time Approximation Schemes*

(EPTAS) where arbitrary functions $f(\varepsilon)$ may only appear as a multiplicative factor but not as exponents of the input size, and *Fully Polynomial Time Approximation Schemes (FPTAS)* with polynomial dependency on $\frac{1}{\varepsilon}$. For a thorough introduction to approximation algorithms and their analysis, we refer to the textbooks by Vazirani [Vaz01] and by Williamson and Shmoys [WS11].

2.3 Scheduling and Packing

Scheduling and packing problems are two of the most fundamental problem classes in combinatorial optimization. In this section, we give a brief introduction into the particular problems we consider in this thesis: machine scheduling and knapsack problems.

2.3.1 Scheduling Problems

Scheduling problems arise in every imaginable aspect of our daily lives whenever we need to assign a set of tasks to scarce resources. Therefore, they have been studied extensively over the last decades. With the rise of new technologies, also the applications and variations of scheduling problems changed over time and have become more ubiquitous. The beginnings of the theoretical analysis of scheduling problems were highly motivated by aspects of production planning arising in economic and industrial applications [AF55, Bel56, Joh54, Wag59], while research today is additionally driven by, e.g., questions appearing in large-scale computing clusters [AKL+15, ALLM18, FBK+12, LMNY13].

In this thesis, we focus on one particular class of scheduling problems, the so-called machine scheduling problems. Here, we are given a set $\mathcal{J}$ of n jobs, i.e., $\mathcal{J} = [n]$, which must be processed by a set of m machines. We usually use the index i to refer to a particular machine if $m > 1$. Each job $j \in \mathcal{J}$ specifies parameters such as its non-negative *processing time* p_j, its *weight* w_j, its *release date* r_j, and its *deadline* d_j. In order to complete, job j needs to be assigned for p_j units of time during the interval $[r_j, d_j)$ to either one or a subset of the machines. This job-to-machine assignment needs to guarantee that no job is processed on several machines at the same time and that no machine is working on more than one job at any given time. We usually refer to such an assignment as *schedule*. Depending on the particular problem, there may be additional constraints that any schedule needs to satisfy in order to be *feasible*. In general, one is interested in selecting a feasible schedule that optimizes some objective function.

In 1979, Graham et al. [GLLRK79a] introduced the *3-field notation* $\alpha \,|\, \beta \,|\, \gamma$ to classify the plethora of different scheduling models. In the following, we use this classification scheme to introduce the building blocks of the scheduling models considered in this thesis.

The first field, α, refers to the machine environment. Single-machine models are represented by $\alpha = 1$. When all jobs can be processed by m parallel, identical machines, we use $\alpha = \mathrm{P}$

to express this. The restriction to scheduling problems with exactly m machines is denoted by $\alpha = \mathrm{P}m$. When jobs can only be processed by some machines and their respective processing times depend on the machine, we denote this by $\alpha = \mathrm{R}$ and use $p_{i,j}$ to indicate for how many time units job j has to be processed if assigned to machine i. We will only consider the *restricted assignment* problem where $p_{i,j} \in \{p_j, \infty\}$ for $p_j \geq 0$.

The second field, β, is used for giving job specific parameters. If jobs can be *preempted*, i.e., interrupted and resumed at a later point, we denote this by $pmtn \in \beta$. In some settings, we distinguish between *migratory* and *non-migratory* preemption where the processing may be resumed on any machine or only on the machine a job was initially started on. In the presence of release dates and deadlines, we sometimes add r_j and d_j, respectively, to the field β as well.

The third field, γ, denotes the particular objective function of the problem. Let C_j refer to the *completion time* of job j in a particular schedule. For minimizing the *makespan*, i.e., the maximal completion time of a job, we use $\gamma = C_{\max}$, where $C_{\max} = \max_j C_j$. Since this problem is equivalent to minimizing the maximal completion time of the machines, we also refer to this problem by *load balancing*. Setting $\gamma = \sum_j C_j$ or $\gamma = \sum_j w_j C_j$ refers to minimizing the sum of completion times or the total weighted completion time, respectively. Let U_j indicate whether job j does not complete on time, i.e., $U_j = 1$ if $C_j > d_j$ and 0 otherwise. Originally, $\gamma = \sum_j w_j U_j$ denotes the objective of minimizing the total weight of jobs completing after their deadline [GLLRK79a]. From an approximation point of view, approximating the minimum of an objective function that can become zero is equivalent to finding the optimum. Hence, we resort to the equivalent maximization problem and denote this by $\gamma = \sum_j w_j(1-U_j)$. Since this objective function asks for maximizing the total weight of jobs completing by their deadline, we also refer to it by weighted *throughput maximization*. The unweighted case is denoted by $\gamma = \sum_j (1 - U_j)$. The last two objective functions differ from the previous ones in the sense that we do not require that each job is scheduled at some point. Instead, we may discard jobs completely at the cost of paying their weight.

The theoretical analysis of scheduling problems dates back several decades and is still growing. As a starting point for a more detailed investigation of scheduling models and algorithms, we refer to the textbook by Pinedo [Pin16] and to the survey articles edited by Leung [Leu04].

2.3.2 Packing Problems

Packing problems describe the task of assigning a set of items to resources with bounded capacities. They arise in a variety of applications in, e.g., logistics, such as in cutting stock, vehicle loading, or pallet packing problems [CKPT17, GG61, Ram92], as well as in computer science, such as placing virtual machines or processes in computing clusters or allocating resources in cloud networks [Sto13, BKB07, BB10]. More applications can be found in the financial sector, e.g., on client level, such as investment selection, or on an institutional level, such as asset-backed securitization and interbank clearing systems [Wei66, GJL98, MP04].

In this thesis, we concentrate on one particular type of packing problems, the Knapsack problem. Here, we are given a set $\mathcal{J}$ of n items, i.e., $\mathcal{J} = [n]$, with sizes $s_j \in \mathbb{N}$ and value $v_j \in \mathbb{N}$ for $j \in \mathcal{J}$. Further, we have one knapsack of capacity S and the task is to find a subset $P \subseteq [n]$ of maximal total value $v(P)$, where $v(P) = \sum_{j \in P} v_j$, such that its total size does not exceed S. The decision variant of this problem is $\mathcal{NP}$-complete and it belongs to the famous list of 21 $\mathcal{NP}$-hard problems by Karp [Kar72]. This problem has been studied extensively since the early days of optimization, which is also reflected by the books on knapsack problems by Martello and Toth [MT90] and by Kellerer, Pferschy, and Pisinger [KPP04]. As observed in the latter, the relevance of this problem is also illustrated by the fact that many important concepts in combinatorial optimization such as approximation schemes and dynamic programming were introduced for or explained by the Knapsack problem.

A straightforward generalization is the Multiple Knapsack problem where there are given m knapsacks with capacities S_i for $i \in [m]$. The task is to select m pairwise disjoint sets $P_i \subseteq [n]$ such as to maximize the total value $\sum_{i=1}^{m} v(P_i)$ while the total size of set P_i does not exceed S_i. In contrast to the Knapsack problem, Multiple Knapsack is strongly $\mathcal{NP}$-hard even for identical knapsack capacities because it generalizes the strongly $\mathcal{NP}$-hard problem 3-Partition [KPP04, GJ79].

Since we also consider scheduling problems, we would like to point out that Multiple Knapsack with m identical knapsacks is equivalent to maximizing the weighted throughput of n jobs with release dates 0, processing times s_j, weights v_j, and identical deadlines S for $j \in [n]$ on m machines, denoted in the 3-field notation by $\mathrm{P} \,|\, d_j = d \,|\, \sum_j w_j(1 - U_j)$.

2.4 Scheduling and Packing Under Uncertainty

This thesis focuses on solving combinatorial optimization problems while dealing with uncertain information. Investigating classical problems under uncertainty is an important step towards bridging the gap between theoretical aspects of optimization that often assumes a simplified view and real-world applications where the future often is unknown. We consider *stochastic information* where the instance is given up-front but certain characteristics, e.g., job processing times or item sizes, are only given as random variables following known probability distributions. The major source of incomplete information in this thesis stems from *online information* where the input is gradually revealed to the optimizer and decisions must be made without complete knowledge about the instance. A closely related concept of uncertain information is *dynamic input* where the input evolves constantly. Elements of the instance, e.g., jobs in scheduling problems or items in packing problems, arrive and depart. In contrast to online optimization, the solution of a dynamic algorithm is allowed to change alongside the instance.

2.4.1 Stochastic Input

In some real-world applications of combinatorial optimization problems, practitioners have some historic information at their disposal which can be used to forecast the future. Based on their expertise, they are able to compile some knowledge about the input such as the job set (in scheduling problems) or the item set (in packing problems) while only the specifics such as processing times or item sizes remain unknown. Using statistical and data-analytical methods, experts can justify knowing the underlying probability distributions of the unknown input.

In the following, we give a brief summary of important notions, concepts, and techniques of probability theory before we formally introduce stochastic scheduling.

Probability theory As we assume that the reader is familiar with basic concepts and techniques of probability theory (and thus with measure theory), we only give a short introduction to the notation used throughout this thesis without formally defining the underlying principles. To this end, let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space. If $E \in \mathcal{F}$, then $\mathbb{P}[E]$ denotes the *probability* that E occurs. Usually, we refer to $E \in \mathcal{F}$ as an *event*. For two events $E, F \in \mathcal{F}$ with $\mathbb{P}[F] > 0$, the probability that E happens under the assumption that F is known to occur is $\mathbb{P}[E \mid F] := \frac{\mathbb{P}[E \cap F]}{\mathbb{P}[F]}$ is; in short, the probability of E *given* F. Two events E and F are *independent* if $\mathbb{P}[E \cap F] = \mathbb{P}[E]\mathbb{P}[F]$. In this case, $\mathbb{P}[E \mid F] = \mathbb{P}[E]$. Intuitively, having information about F does not increase the information about E.

Let $X : \Omega \to \mathbb{R}$ be a real-valued random variable. If X follows the probability distribution D, we say $X \sim D$. Moreover, $\mathbb{E}[X]$ denotes the *expected value* and $\mathbb{V}\mathrm{ar}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$ denotes the *variance* of a random variable X. Sometimes we are also interested in $\mathrm{SCV}[X] = \frac{\mathbb{V}\mathrm{ar}[X]}{\mathbb{E}[X]^2}$, the *squared coefficient of variation*.

Two random variables $X, Y : \Omega \to \mathbb{R}$ are *independent* if, for all measurable sets $I, I' \subseteq \mathbb{R}$, it holds that

$$\mathbb{P}[\{X \in I\} \cap \{Y \in I'\}] = \mathbb{P}[X \in I]\mathbb{P}[Y \in I'].$$

Intuitively, independence of X and Y implies that having knowledge about the realization of X does not increase the information about the realization of Y.

In the following, we give some important properties of random variables. Since these are commonly known results from probability theory, we refer to book (chapters) for proofs.

The Law of Total Expectation gives us a way of calculating the expected value of X by evaluating $\mathbb{E}[X]$ separately on a set of disjoint events. A proof can be found in the book by Biagini and Campanino [BC16].

Theorem 2.1 (Law of Total Expectation). *Let $\{E_i\}_{i \in I}$ be a countable partition of Ω. Then,*

$$\mathbb{E}[X] = \sum_{i \in I} \mathbb{E}[X \mid E_i]\mathbb{P}[E_i].$$

The following inequality is known as Markov's Inequality. A proof for this famous inequality can be found in Chapter 3.1 of the book by Gut [Gut13].

Theorem 2.2 (Markov's Inequality). *Suppose that* $\mathbb{E}[|X|] < \infty$ *for a random variable* X*. Then,*

$$\mathbb{P}[|X| > x] \leq \frac{\mathbb{E}[|X|]}{x}.$$

The following concentration result for independent random variables distributed in $[0,1]$ is a very useful variant of the Chernoff-Hoeffding bound. A proof can be found in the book by Dubhashi and Panconesi [DP09].

Theorem 2.3 (Chernoff-Hoeffding Bound). *For* $j \in [n]$*, let* X_j *be independently distributed in* $[0,1]$ *and let* $X := \sum_{j=1}^{n} X_j$*. For* $0 < \varepsilon < 1$*, it holds that*

(i) $\mathbb{P}[X \geq (1+\varepsilon)\mathbb{E}[X]] \leq \exp\left(-\varepsilon^2 \frac{\mathbb{E}[X]}{3}\right)$ *and*

(ii) $\mathbb{P}[X \leq (1-\varepsilon)\mathbb{E}[X]] \leq \exp\left(-\varepsilon^2 \frac{\mathbb{E}[X]}{2}\right)$,

where $\exp(x)$ *denotes the exponential function of* x*.*

For a deeper introduction to probability theory, we refer the reader to some introductory textbooks such as the one by Gut [Gut13] or the one by Biagini and Campanino [BC16]. Since discrete probability distributions play a major role in those chapters of this thesis dealing with stochastic information, we recommend the book by Johnson, Kemp, and Kotz [JKK05] and the references therein.

The two main discrete distributions occurring in this thesis are *Bernoulli* and *binomial* distributions. We say that a random variable X follows a Bernoulli distribution with success probability $q \in [0,1]$ or, short, $X \sim \mathrm{Ber}(q)$, if

$$\mathbb{P}[X = 1] = q = 1 - \mathbb{P}[X = 0].$$

For a Bernoulli random variable it holds that $\mathbb{E}[X] = q$ and $\mathbb{V}\mathrm{ar}[X] = q(1-q)$.

A random variable X is binomially distributed with success probability $q \in [0,1]$ and size parameter $n \in \mathbb{N}$ or, short, $X \sim \mathrm{Bin}(n,q)$, if

$$\mathbb{P}[X = k] = \binom{n}{k} q^k (1-q)^{n-k},$$

for $k \in \{0, 1, \ldots, n\}$. If $X \sim \mathrm{Ber}(n,q)$, then $\mathbb{E}[X] = nq$ and $\mathbb{V}\mathrm{ar}[X] = nq(1-q)$. If X_j, for $j \in [n]$, are n independent $\mathrm{Ber}(q)$-distributed random variables, then $X := \sum_{j=1}^{n} X_j$ follows a $\mathrm{Bin}(n,q)$ distribution.

Stochastic scheduling In stochastic scheduling, the uncertainty in the input is modeled via random processing times. More precisely, we are given a set $\mathcal{J}$ of n jobs whose processing times P_j are random variables for $j \in \mathcal{J}$. We assume complete knowledge about the distribution of the random processing time $P_j \geq 0$ and use p_j to refer to a particular realization of P_j. For two jobs, we impose independence on their processing times. This restriction is not inherent in stochastic scheduling but extremely helpful due to certain methods borrowed from probability theory. Other job characteristics such as release dates or deadlines are known in advance, which is also the case for the machine environment.

As the processing times are not deterministic anymore, the solution to a problem is not a schedule but a so-called *scheduling policy*. A scheduling policy decides in an "online" matter which jobs to schedule next on which machine. Here, the "online" nature of the problem lies in the fact that the scheduling policy obtains information about the instance by scheduling jobs and by observing how the conditional distributions of the processing times of currently processed jobs evolve. Then, these observations and the a priori knowledge about the instance guide the decision process of the scheduler.

This intuition is made more precise by Möhring, Radermacher, and Weiss [MRW84,MRW85]: a scheduling policy Π specifies a set of possible *actions* at *decision times* t. An action comprises a set of jobs to start at time t and the next (tentative) decision time t'. If an action is taken at time t, the next decision has to be made at time t' or upon a job's release or completion at time $t'' < t'$. The decision for a certain action at time t only depends on the t-*past* of the realization, i.e., the information observed up to time t. This information consists of the realized processing times of jobs already completed at time t and the conditional distributions of the jobs started before time t but not yet completed. Such a policy is called *non-anticipatory*. A particular class of non-anticipatory policies are the *elementary* policies where decisions only happen upon release or completion of a job.

Then, C_j^Π, the completion time of job j under scheduling policy Π, is a random variable depending on Π as well as the realization of the processing times. Since simple examples already show that a point-wise optimal scheduling policy, i.e., optimal for each realization, does in general not exist [MR85], we are interested in minimizing the cost function in expectation. That is, for an instance $\mathcal{I}$, we are interested in finding an *optimal* policy Π^* with

$$\mathbb{E}[\, f(\Pi^*, \mathcal{I})\,] = \min\Big\{\mathbb{E}[\, f(\Pi, \mathcal{I})\,] : \Pi \text{ non-anticipatory scheduling policy}\Big\},$$

where $f(\Pi, \mathcal{I})$ denotes the (random) objective function value for instance $\mathcal{I}$ under policy Π. We emphasize that the scheduling policies considered in this thesis can be *adaptive* in the sense that the set of actions at time t is allowed to depend on the t-past. Further below we briefly discuss the class of *non-adaptive* scheduling policies.

It has been shown that for some processing time distributions and some objective functions such an optimal policy does exist and can easily be expressed [Gla79,BDF81,WVW86,Rot66].

However, as Pinedo and Weiss [PW87] show, this optimal policy is hard to describe for general processing time distributions. Therefore, we resort to *approximate policies* as introduced by Möhring, Schulz, and Uetz [MSU99] that are closely related to approximation algorithms previously discussed. Due to the stochastic nature of the underlying problem, we only require that the decision of a policy can be computed in polynomial time while, there is no such bound on the time horizon of the schedule itself.

Definition 2.4 (α-approximate policies). *Let $\alpha \geq 1$. A scheduling policy Π that can be computed in polynomial time is an* α-approximate *policy if*

$$\mathbb{E}[\, f(\Pi, \mathcal{I}) \,] \leq \alpha \mathbb{E}[\, f(\Pi^*, \mathcal{I}) \,]$$

holds for all instances $\mathcal{I}$, where Π^* denotes an optimal policy for instance $\mathcal{I}$. We also use α-approximation *to refer to such a policy Π. The infimum α such that Π is an α-approximation is called* approximation factor *or* approximation ratio.

Not requiring a bound on the time horizon is mostly due to the fact that we consider random variables with (possibly) exponential or even unbounded support. In particular, already for a single job on a single machine there might be a non-zero probability that the realization of its processing time is exponentially larger than indicated by, e.g., its expected value. Hence, for some realizations a polynomial encoding might not be possible. As discussed by Skutella, Sviridenko, and Uetz [SSU16], it is not likely that such a situation occurs due to Markov's Inequality (Theorem 2.2), but one should be aware of this possibility.

Most results on approximation policies consider the problem of minimizing the total weighted completion time and variations thereof while the approximation guarantee mostly depends linearly on Δ [MSU99, SU05, SSU16, MUV06, GMUX20, JS18]. Here, Δ is an upper bound on the squared coefficients of variation of the processing times. An exception worth mentioning is the $\mathcal{O}(\log^2 n + m \log n)$-approximate policy by Im, Moseley, and Pruhs [IMP15] for minimizing the expected weighted total completion time.

When minimizing the makespan on parallel machines, approximation becomes easier: It is common knowledge that List Scheduling is already $\left(2 - \frac{1}{m}\right)$-approximate. In fact, this approximation guarantee even holds per realization.

Non-adaptive scheduling A line of work that is orthogonal to the adaptive setting is the non-adaptive model. The assignment of jobs to machines happens upfront before the randomness is revealed. Hence, the non-adaptive model is sometimes also called *fixed assignment*. Since approximation algorithms are again evaluated relative to an optimal algorithm of the same nature, the adaptive and the non-adaptive models are mutually incomparable.

When minimizing the total completion time objective, this model immediately reduces to its deterministic counterpart by linearity of expectation. Any result obtained there transfers

to the stochastic setting by using $\mathbb{E}[P_j]$ as deterministic surrogate for the processing time of job j. As shown by Skutella, Sviridenko, and Uetz [SSU16], the adaptivity gap for minimizing the weighted completion time on identical machines is $\Omega(\Delta)$. That is, the ratio between the best fixed-assignment policy and the optimal adaptive scheduling policy is at least $\Omega(\Delta)$. Nevertheless, some of the results in the adaptive setting use fixed machine assignments [SSU16, GMUX20, MUV06]. This indicates that fixed-assignment policies are seemingly more tractable for this particular objective function.

Conversely, makespan minimization seems to be the more difficult objective function in the non-adaptive setting. Kleinberg, Rabani, and Tardos [KRT00] were the first to obtain constant approximation ratios for parallel machines. Recently, Gupta et al. [GKNS18] were able to obtain the first constant approximation guarantee for load balancing on unrelated machines. They also looked into the problem of minimizing the q-norm of the load vector and obtained the first non-trivial approximation guarantee of $\mathcal{O}\left(\frac{q}{\log q}\right)$. This result was later improved to a constant factor approximation by Molinaro [Mol19]. Gupta et al. [GKNS18] showed an adaptivity gap of $\Omega\left(\frac{\log m}{\log \log m}\right)$ for makespan minimization even for identical jobs on parallel identical machines.

Stochastic packing problems Here, the item sizes follow independent random variables, and a subset or the complete set of items has to be packed to optimize some objective function subject to some constraints. Such a constraint could be the satisfaction of a bound on the overflow probability, i.e., the probability to exceed the capacity of a bin or a knapsack. For special probability distributions, Goel and Indyk [GI99] derive constant approximation guarantees for Stochastic Bin Packing and Stochastic Knapsack. Kleinberg, Rabani, and Tardos [KRT00] consider special distributions and relax either the capacity constraint or the bound on the overflow probability. Dean, Goemans, and Vondrak [DGV08] consider another variant of Stochastic Knapsack: The items have to be packed into the knapsack following an adaptive or a non-adaptive order, and upon placing an item, its size is realized. The goal is to pack as much value as possible before the first item "overflows" the knapsack. They derive $\mathcal{O}(1)$-approximate policies in both models. Interestingly, the value attained by their non-adaptive policy is within a constant factor of the optimal adaptive policy.

2.4.2 Online Input

In the *online* model, the instance is revealed only incrementally and the optimizer has to take irreversible actions without complete knowledge about the future. From an algorithmic point of view, this implies that, no matter the future input, the taken actions can either not be undone anymore or only at an extremely high cost which then influences the overall performance of the online algorithm. Conversely, an *offline* algorithm has access to the entire instance in advance and bases its decisions on this complete knowledge.

As observed by Borodin and El-Yaniv [BE98], some problems, such as scheduling and packing, are meaningful and natural in the offline and the online world: In one's own computing cluster, the workload usually is known and can be scheduled accordingly, while a cloud computing provider does not know the computing requests before their submission to the system. For a logistics company, the delivery of goods can be planned according to the trucks' capacities, while the pickup of goods possibly leads to decision making "on the fly" or, in other words, online. Other problems, such as paging or routing packets in a computer network, are intrinsically online, and asking for an offline algorithm neglects a major part of the problem.

For a thorough introduction to online algorithms, their applications, and analysis, we refer to the book by Borodin and El-Yaniv [BE98], the collection of surveys by Fiat and Woeginger [FW98], and the survey by Albers [Alb03]. The surveys by Sgall [Sga96] and by Pruhs, Sgall, and Torng [PST04] put a special emphasis on online scheduling algorithms. Regarding the way new information is released, Pruhs, Sgall, and Torng [PST04] coined two different terms, which we now briefly summarize.

Online-list model Under this paradigm, the unknown elements are ordered in some list; hence the term *online-list model*. Once the element is revealed, the algorithm knows all its characteristics, such as size or processing time. Based on the information seen so far, the algorithm has to deal with the element before the next element is observed. After the decision how to deal with the newly revealed element is made, the algorithm is not allowed to change or revoke it. In scheduling problems, this implies that a job has to be scheduled on a machine (satisfying the specific problem requirements) without knowledge about the remaining job set. This variant models, e.g., load balancing decisions in computing clusters. Since usually there are no time horizons in packing problems, their online counterparts are mostly modeled in this setting as well.

Online-time model In contrast to the online-list model, the *online-time model* also takes into account the time between the assignment of jobs. That is, jobs *arrive* online over time at their release dates which is when their characteristics become known to the online algorithm. Hence, the time horizon itself plays a major role in this paradigm and has to be taken into account when decisions are made. This implies that the scheduler can sometimes delay decisions at a certain problem-specific cost. Depending on the particular problem, the algorithm can alter the current schedule in favor of newly arrived jobs. This model covers admission decisions for computing clusters since the requests are typically submitted over time at the point most convenient for the client.

Competitive analysis Since an online algorithm cannot base its decision on complete knowledge of the instance, it typically cannot find an optimal solution. To be able to evaluate a particular algorithm or compare two with each other, we can use a similar approach as for

approximation algorithms: worst-case analysis where the performance of an algorithm is compared to the optimal solution. In the context of online algorithms, the optimal solution has complete knowledge about the instance, and hence is called *offline optimum*, while an online algorithm gains access to this knowledge depending on the particular model. Sleator and Tarjan [ST85] introduced the term *competitive analysis* for this method of worst-case analysis.

Let $\mathcal{A}$ denote a particular online algorithm for a given problem and let $f(\mathcal{A}, \mathcal{I})$ denote the objective function value achieved by $\mathcal{A}$ for instance $\mathcal{I}$. Similar to the notion of approximation algorithms, the definition of a *competitive* algorithm depends on the optimization goal, i.e., it depends on the problem being a minimization or a maximization problem.

Definition 2.5 (c-competitive algorithm). *Let $c \geq 1$. An online algorithm $\mathcal{A}$ is* c-competitive *for a minimization problem if, for all instances $\mathcal{I}$,*

$$f(\mathcal{A}, \mathcal{I}) \leq c \cdot f^*(\mathcal{I}),$$

where $f^*(\mathcal{I})$ denotes the objective function value of an offline optimum for instance $\mathcal{I}$. For a maximization problem, a c-competitive algorithm must satisfy, for all instances $\mathcal{I}$

$$f(\mathcal{A}, \mathcal{I}) \geq \frac{1}{c} f^*(\mathcal{I}).$$

If c is the infimum such that $\mathcal{A}$ is c-competitive, then c is called the competitive ratio *of $\mathcal{A}$.*

We emphasize that competitiveness of an online algorithm does not imply any bounds on the computational power since the key difficulty is decision making under uncertainty and strict irrevocability constraints. Nevertheless, many known algorithms do run in polynomial time.

Migration and recourse For some problems, there exist strong lower bounds on the competitive ratio of online algorithms that are not allowed to revoke decisions. That is, it can be shown that the lack of information prevents every online algorithm for such a problem from having a competitive ratio less than c. For many problems, the irrevocability assumption of online algorithms is overly pessimistic, and in some applications changes are rather sensible as long as these changes are bounded in some way. To address these issues, concepts to soften the irrevocability requirement have been developed. There are two major streams towards more adaptive models with a softened irrevocability requirement: online optimization with *recourse* and with *migration*. Both models permit an online algorithm to change previously made decisions upon gaining more knowledge about the instance. Of course, if these changes were not bounded in some way, the online nature of the problem would be lost since an algorithm could simulate the current offline optimum. One can enforce such a bound in average over the first k arrivals or per round, leading to an *amortized* or *non-amortized* bound on the changes.

With recourse, we refer to the possibility to change decisions under the requirement that the *number* of such changes remains bounded [MSVW16, IW91, GGK16, GKS14]. This model

may be employed when the underlying change in a solution is negligible, e.g., when the cost of reassigning an item does not depend on the size of the item that has to be moved.

The term migration is used to refer to a bound on the *volume* of changes [SSS09,SV16,JK19]. That is, the cost for repacking an item or rescheduling a job is proportional to their respective size. This models the fact that a change in the solution might lead to the need for physically adapting the assignment which in turn is cheaper if the element in question is small.

The model that we consider in this thesis is a generalization of the former two: bounding the *reassignment cost* [AGZ99, Wes00]. Specifically, each element comes with an individual (re)assignment cost that needs to be paid upon (re)assigning the element and the task is to maintain a good solution with moderate reassignment cost. Since we only consider scheduling and packing problems, we use the term "reassignment" to refer to scheduling a job on another machine or packing an item in another knapsack. By choosing uniform reassignment costs or setting them equal to the size of the respective element, we recover online optimization with recourse or migration, respectively.

2.4.3 Dynamically Changing Input

A seemingly related concept that deals with uncertain input is that of *dynamic* algorithms. An algorithm is said to be dynamic if it maintains a solution to a certain problem even if the instance undergoes small modifications in each round. For packing problems, these modifications may include the arrival or departure of an item or a knapsack/bin. Similarly, for scheduling problems, the instance may be modified by removing or inserting a job or a machine as well as by changing the weight of a job. The task is to maintain a good solution in each round while spending only little computation time per round. Depending on the hardness of the underlying static problem, "good" refers either to optimal or near-optimal solutions.

The dynamic algorithm can be seen as a data structure that efficiently supports updates, i.e., modifications of the instance, and that is then, given these updates, used to construct a solution if queried. That is, *explicit* solutions that allow for linear access time of a solution are not required. Instead, only *implicit* solutions are maintained that answer certain queries sufficiently fast. Usually, such a query asks for the output of the entire solution or for the status of a certain element in the solution, e.g., for the knapsack in which the queried element is packed or the machine on which the queried job is processed. This relaxation on the representation of a solution allows for a trade-off between the update and the query time. Hence, when comparing dynamic algorithms to traditional algorithms that compute an explicit solution, one should be aware of this difference.

Since the motivation behind dynamic algorithms are applications with mostly local changes, we would like to use the solution computed prior to the update in order to obtain a new solution significantly faster than when starting from scratch. Further, the new solution should reflect the changes made to the instance and is thus allowed to change as well. For example in scheduling

problems, a noticeable amount of jobs can be assigned to another machine in each round as long as the reassignment can be computed, not executed, sufficiently fast. We emphasize this difference to the previously introduced models of online optimization with reassignment.

Typically, dynamic algorithms are investigated in the context of graph problems, and we refer to the surveys [DEGI10, Hen18, BP11] for an overview on dynamic graph algorithms. For some connectivity problems [HK99, HdLT01], such as Minimum Spanning Tree or 2-Edge Connectivity, and for Vertex Cover [BHN17, BK19], there are dynamic algorithms with poly-logarithmic update time although the majority of graph problems did not seem to allow for fast algorithms. Only recently, researchers started to investigate the reasons for this lack of efficient algorithms and proved conditional lower bounds; see, e.g., [AW14]. Set Cover admits near-optimal approximation algorithms with poly-logarithmic update times and has been studied extensively [BHN19, BHI15, GKKP17, AAG+19].

There is little research on efficient near-optimal algorithms for scheduling or packing problems. A notable exception is a $\frac{5}{4}$-approximate algorithm with poly-logarithmic update time for Bin Packing by Ivkovic and Lloyd [IL98]. For a detailed introduction to dynamic algorithms in the context of combinatorial optimization problems, we refer to the survey by Boria and Paschos [BP11].

3

Stochastic Minsum Scheduling

Minimizing the sum of completion times when scheduling jobs on m identical parallel machines is a fundamental scheduling problem. Unlike the well-understood deterministic variant, it is a major open problem how to handle stochastic processing times. We show for the prominent class of index policies that no such policy can achieve a distribution-independent approximation factor. This strong lower bound holds even for simple instances with only deterministic jobs of uniform size and identically two-point distributed stochastic jobs. For such instances, we give an $\mathcal{O}(m)$-approximate list scheduling policy. Moreover, we derive further bounds on the instance parameters that allow for $\mathcal{O}(1)$-approximate list scheduling policies.

Bibliographic Remark: Parts of this chapter are joint work with F. Fischer, J. Matuschke, and N. Megow and correspond to or are identical with [EFMM19].

Table of Contents

3.1 Introduction

Scheduling jobs on identical parallel machines with the objective to minimize the sum of completion times is a classical and well-understood problem. Here, we are given a set $\mathcal{J}$ of n jobs, where each job $j \in \mathcal{J}$ has a processing time p_j that indicates for how many time units it has to be processed non-preemptively on one of the m given machines. At any point in time, a machine can process at most one job. The objective is to find a schedule that minimizes the total completion time, $\sum_{j\in\mathcal{J}} C_j$, where C_j denotes the completion time of job j. This problem is denoted by $\mathrm{P}||\sum C_j$ in the standard three-field notation [GLLRK79a]. It is well known that scheduling the jobs as early as possible in Shortest Processing Time (SPT) order solves the problem optimally on a single [Smi56] as well as on multiple machines [CMM67].

Stochastic scheduling Uncertainty in the processing times is ubiquitous in many applications. Although the first results on scheduling with probabilistic information date back to the 1960s, the question how to schedule jobs with stochastic processing times is hardly understood.

We investigate a stochastic variant of the minsum scheduling problem. The processing time of a job j is modeled by a random variable P_j with known probability distribution. We assume that the processing time distributions for individual jobs are independent. The objective is to find a *non-anticipatory* scheduling policy Π that decides for any time t, with $t \geq 0$, which jobs to schedule. A non-anticipatory policy may base these scheduling decisions only on observed information up to time t and a priori knowledge about the distributions. In particular, the policy is not allowed to use information about the actual realizations of processing times of jobs that have not yet started by time t. For a more in-depth introduction to non-anticipatory scheduling policies we refer to Section 2.4.1.

For a non-anticipatory policy Π, the value of the objective function $\sum_j C_j^\Pi$ is a random variable. A natural generalization of the deterministic problem $\mathrm{P}||\sum C_j$ is to ask for minimizing the *expected value* of this random variable, i.e., to minimize $\mathbb{E}\left[\sum C_j^\Pi\right]$, where the expectation is taken over the randomness in the processing time variables. We drop the superscript whenever the policy is clear from the context. This stochastic scheduling problem is denoted by $\mathrm{P}|\,|\mathbb{E}[\sum C_j]$.

List scheduling and index policies An important class of policies in (stochastic) scheduling is List Scheduling as defined by Graham [Gra69]. A List Scheduling policy maintains a (static or dynamic) priority list of jobs and schedules at any time as many available jobs as possible in the order given by the list. The aforementioned SPT rule falls into this class. List scheduling policies are the simplest type of *elementary* policies, that is, policies that start jobs only at the completion times of other jobs (or at time 0). For further details on the classification of (non-preemptive) stochastic scheduling policies, we refer to the work of

Möhring, Radermacher, and Weiss [MRW84, MRW85].

A prominent subclass of list scheduling policies is called *index policies* [Git89, Wal88]. An index policy assigns a priority index to each unfinished job, where the index for a job is determined by the (distributional) parameters and the current state of execution of the job itself but independent of other jobs. If job preemption is not allowed, then these priority indices are static, that is, they do not change throughout the execution of the scheduling policy. Moreover, index policies assign to jobs with the same probability distribution the same priority index and do not take the number of jobs or the number of machines into account.

In the first paper on stochastic processing times [Rot66], Rothkopf showed that scheduling the jobs in Weighted Shortest Expected Processing Time (WSEPT) order, i.e., in non-increasing order of $\frac{w_j}{\mathbb{E}[P_j]}$, is optimal for minimizing the total expected weighted completion time on one machine. If the processing times follow a geometrical distribution, Glazebrook [Gla79] showed that List Scheduling in order of Shortest Expected Processing Time (SEPT) is optimal for minimizing $\sum_j \mathbb{E}[C_j]$ on parallel identical machines. For exponentially distributed processing times, SEPT is also optimal according to Bruno, Downey, and Frederickson [BDF81]. Weber, Varaiya, and Walrand [WVW86] generalize these results to instances where the processing times are totally ordered stochastically. That is, for every two jobs $j, k \in \mathcal{J}$, their processing times P_j and P_k are *stochastically comparable*, meaning that either $\mathbb{P}[P_j > x] \leq \mathbb{P}[P_k > x]$ or $\mathbb{P}[P_k > x] \leq \mathbb{P}[P_j > x]$ for all $x \in \mathbb{R}$.

Other index policies that perform provably well for certain stochastic scheduling settings are, e.g., List Scheduling in Longest Expected Processing Time (LEPT) order as shown by Weber [Web82], the Largest Variance First (LVF) rule as observed by Pinedo and Weiss [PW87], and the *Gittins Index* [Git79]. For an overview on theory and applications of index policies (with a focus on interruptible jobs) we refer to the works by Gittins, Glazebrook, and Weber [GGW11] and by Glazebrook et al. [GHKM14].

Further related results For arbitrary instances of $\mathrm{P}|\,|\mathbb{E}[\sum C_j]$, no optimal policy is known. Thus, research focuses on approximate policies. Starting with the seminal paper by Möhring, Schulz, and Uetz [MSU99], several scheduling policies were analyzed for this problem (with arbitrary job weights) and generalizations, such as precedence constraints [SU05], heterogeneous machines [GMUX17, SSU16], and online models [GMUX17, MUV06, Sch08]. In all cases, the approximation guarantee depends on the probability distributions of the processing times. More precisely, the guarantee is in the order $\mathcal{O}(\Delta)$, where Δ is an upper bound on the squared coefficients of variation of the processing time distributions P_j, that is, $\frac{\mathbb{V}\mathrm{ar}[P_j]}{\mathbb{E}[P_j]^2} \leq \Delta$ for all j.

Besides linear programming relaxations, the (W)SEPT policy plays a key role in the aforementioned results. This index policy, being optimal on a single machine, has been studied extensively as a promising candidate for solving $\mathrm{P}|\,|\mathbb{E}[\sum C_j]$ with bounded approximation ratio. Recently, the upper bound for WSEPT has been decreased to $\frac{1+(\sqrt{2}-1)}{2}(1+\Delta)$ by Jäger and Skutella [JS18]. On the negative side, it has been shown independently that neither

WSEPT [Lab13] nor SEPT [CFMM14,IMP15] can achieve approximation factors independent of Δ when there are non-constantly many machines.

A remarkable result is a List Scheduling policy for $P|\,|\mathbb{E}[\sum C_j]$ with the first distribution-independent approximation factor of $\mathcal{O}(m \log n + \log^2 n)$ by Im, Moseley, and Pruhs [IMP15]. This policy is based on SEPT, but in addition, it carefully takes into account the probability that a job turns out to be long.

Nevertheless, it remains a major open question whether there is a constant factor approximation for this problem even if all weights are equal. Interestingly, there is an index policy with an approximation factor 2 for the *preemptive* (weighted) variant of our stochastic scheduling problem by Megow and Vredeveld [MV14]. It is natural to ask whether index policies can achieve a constant approximation factor also in the non-preemptive setting.

Our contribution As our main result, we rule out any constant or even distribution-independent approximation ratio for index policies. More precisely, we give a lower bound of $\Omega(\Delta^{1/4})$ for the approximation ratio of any index policy for $P|\,|\mathbb{E}[\sum C_j]$. This strong lower bound implies that prioritizing jobs only according to their individual processing time distributions cannot lead to better approximation ratios. More sophisticated policies are needed that take the entire job set and the machine setting into account. Somewhat surprisingly, our lower bound holds for very simple instances with only two types of jobs, deterministic jobs of uniform size and stochastic jobs that all follow the same two-point distribution. For this class of instances, we provide an alternative list scheduling policy — carefully taking into account the number of jobs and machines — and show that it is an $\mathcal{O}(m)$-approximation. If the deterministic jobs are identical, we obtain constant approximation ratios for certain combinations of parameters.

3.2 Lower Bound for Index Policies

In this section, we prove our main result, a distribution-dependent lower bound on the approximation factor of any index policy.

Theorem 3.1. *Any index policy has an approximation factor* $\Omega(\Delta^{1/4})$ *for* $P||\mathbb{E}[\sum_j C_j]$.

To prove this lower bound, we consider a simple class of instances that we call *Bernoulli-type instances*. This class consists of two types of jobs, deterministic jobs $\mathcal{J}_d$ and stochastic jobs $\mathcal{J}_s$. A job $j \in \mathcal{J}_d$ has deterministic processing time p_j while a job $j \in \mathcal{J}_s$ has processing time 0 with probability $q \in (0,1)$ and l with probability $1-q$, where $l > 0$. Let $n_d = |\mathcal{J}_d|$ and $n_s = |\mathcal{J}_s|$.

For the stochastic jobs, i.e., $j \in \mathcal{J}_s$, let $X_j = \mathbb{1}_{\{P_j = l\}}$. That is, X_j is a Bernoulli-distributed random variable that indicates whether or not $j \in \mathcal{J}_s$ is *long*, i.e., $P_j = l$. Let $X = \sum_{j \in \mathcal{J}_s} X_j$. Since the processing time variables P_j are independent, the same holds for $X_j, j \in \mathcal{J}_s$. Hence, X follows a binomial distribution with success probability q, size parameter n_s, and expected value $n_s \cdot q$. In other words, X counts the number of jobs that turn out to be long.

Proof of Theorem 3.1. Let $\Delta > 0$; it will act as an upper bound on the squared coefficients of variation. We define two families of Bernoulli-type instances, $\mathcal{I}_1(\Delta, m)$ and $\mathcal{I}_2(\Delta, m)$. The instances differ only in the number of deterministic and stochastic jobs but not in the processing time distributions. We define the processing time for deterministic jobs in $\mathcal{J}_d$ to be equal to 1, i.e., $p_j = 1$ if $j \in \mathcal{J}_d$, and for stochastic jobs $j \in \mathcal{J}_s$ we define

$$P_j = \begin{cases} 0 & \text{with probability } 1 - \frac{1}{\Delta} \\ \Delta^{3/2} & \text{with probability } \frac{1}{\Delta}. \end{cases}$$

Note that $\mathbb{E}[\,P_j\,] = \Delta^{1/2}$ and $\mathbb{V}ar[P_j] = \Delta^2 - 1$ for $j \in \mathcal{J}_s$. Hence, the squared coefficients of variation are at most Δ.

For such Bernoulli-type instances, there are only two index policies, one where the deterministic jobs have higher priority, denoted by $\mathcal{J}_d \prec \mathcal{J}_s$, and one where the stochastic jobs have higher priority, denoted by $\mathcal{J}_s \prec \mathcal{J}_d$. We show that for any fixed $\Delta > 1$, there exists a value of m such that the cost of the schedule produced by $\mathcal{J}_d \prec \mathcal{J}_s$ on instance $\mathcal{I}_1(\Delta, m)$ is greater by a factor of $\Omega(\Delta^{1/4})$ than the cost of the schedule produced by $\mathcal{J}_s \prec \mathcal{J}_d$ and vice versa for instance $\mathcal{I}_2(\Delta, m)$. Since the instances $\mathcal{I}_1(\Delta, m)$ and $\mathcal{I}_2(\Delta, m)$ are indistinguishable by an index policy, this result implies the lower bound.

Instance $\mathcal{I}_1(\Delta, m)$ Instance $\mathcal{I}_1(\Delta, m)$ is defined by letting $n_d = \Delta^{3/4}m$ and $n_s = \frac{1}{2}\Delta m$; without loss of generality we assume $\frac{n_d}{m} \in \mathbb{Z}$. We distinguish both priority orders.

Deterministic jobs before stochastic jobs When the deterministic jobs are scheduled first, no job in $\mathcal{J}_s$ starts before time $\frac{n_d}{m}$. Thus,

$$\mathbb{E}\left[\sum_{j \in \mathcal{J}} C_j\right] \geq \frac{n_d}{m} n_s = \frac{1}{2}\Delta^{7/4} m.$$

Stochastic jobs before deterministic jobs Let X be the random variable counting the number of jobs in $\mathcal{J}_s$ that turn out to be long. That is, $X \sim \mathrm{Bin}(n_s, \frac{1}{\Delta})$ and $\mathbb{E}[\,X\,] = \frac{m}{2}$. We distinguish two cases based on the value of X.

If $X \leq \frac{3}{4}m$, every stochastic job starts at time 0. Hence,

$$\mathbb{E}\left[\sum_{j \in \mathcal{J}_s} C_j \,\middle|\, X \leq \frac{3}{4}m\right] \leq \frac{3}{4}\Delta^{3/2} m.$$

At least $\frac{m}{4}$ machines are free for scheduling deterministic jobs, $\mathcal{J}_d$, at total cost bounded by

$$\mathbb{E}\left[\sum_{j \in \mathcal{J}_d} C_j \,\middle|\, X \leq \frac{3}{4}m\right] \leq \frac{n_d(n_d+1)}{\frac{1}{4}m} \leq 8\Delta^{3/2} m.$$

In the case $X > \frac{3}{4}m$, we get a (very crude) upper bound on the expected cost by assuming that all jobs have processing time $\Delta^{3/2}$ and then scheduling them on a single machine:

$$\mathbb{E}\left[\sum_{j\in\mathcal{J}} C_j \,\middle|\, X > \frac{3}{4}m\right] < \frac{1}{2}(n_d+n_s)(n_d+n_s+1)\Delta^{3/2} \leq 3\Delta^{7/2}m^2.$$

By the Chernoff-Hoeffding bound (Theorem 2.3), the probability of the second case is at most $\exp\left(\frac{-m}{24}\right)$. Using the Law of Total Expectation (Theorem 2.1),

$$\begin{aligned}\mathbb{E}\left[\sum_{j\in\mathcal{J}} C_j\right] &\leq \mathbb{P}\left[X \leq \frac{3}{4}m\right]\mathbb{E}\left[\sum_{j\in\mathcal{J}} C_j \,\middle|\, X \leq \frac{3}{4}m\right] + \mathbb{P}\left[X > \frac{3}{4}m\right]\mathbb{E}\left[\sum_{j\in\mathcal{J}} C_j \,\middle|\, X > \frac{3}{4}m\right] \\ &\leq \frac{3}{4}\Delta^{3/2}m + 8\Delta^{3/2}m + \exp\left(-\frac{m}{24}\right)\cdot 3\Delta^{7/2}m^2 \in \mathcal{O}(\Delta^{3/2}m)\end{aligned}$$

for sufficiently large m. Thus, on sufficiently many machines, the index policy $\mathcal{J}_d \prec \mathcal{J}_s$ has total cost greater by a factor of $\Omega(\Delta^{1/4})$ than that of policy $\mathcal{J}_s \prec \mathcal{J}_d$.

Instance $\mathcal{I}_2(\Delta, m)$ Instance $\mathcal{I}_2(\Delta, m)$ is defined by $n_d = \Delta^{5/4}m$ and $n_s = 2\Delta m$; we assume without loss of generality $\frac{n_d}{m} \in \mathbb{Z}$. Let X denote again the number of jobs in $\mathcal{J}_s$ that turn out to be long, i.e., $X \sim \mathrm{Bin}(2\Delta m, \frac{1}{\Delta})$ and hence, $\mathbb{E}[\,X\,] = 2m$. We analyze both index policies.

Deterministic jobs before stochastic jobs We condition on two disjoint events regarding the realized value of X. If $X \leq 3m$, every (stochastic) job has completed by time $\Delta^{5/4} + 3\Delta^{3/2}$. Further, the cost of $\mathcal{J}_d \prec \mathcal{J}_s$ is upper bounded by the that of the following policy: Assign to every machine at most $\frac{n_d}{m} = \Delta^{5/4}$ deterministic jobs and at most three long stochastic jobs. Thus,

$$\mathbb{E}\left[\sum_{j\in\mathcal{J}} C_j \,\middle|\, X \leq 3m\right] \leq \frac{n_d^2}{m} + \left(\Delta^{5/4} + 3\Delta^{3/2}\right) n_s \in O(\Delta^{5/2}m)$$

The case $X > 3m$ happens with probability at most $\exp\left(\frac{-m}{6}\right)$ by the Chernoff-Hoeffding bound (Theorem 2.3). Using again the fact that scheduling all jobs on one machine and assuming $P_j = \Delta^{3/2}$ for $j \in \mathcal{J}$ yields an upper bound,

$$\mathbb{E}\left[\sum_{j\in\mathcal{J}} C_j \,\middle|\, X > 3m\right] \leq 3\Delta^{7/2}m^2.$$

With the Law of Total Expectation (Theorem 2.1),

$$\mathbb{E}\left[\sum_{j\in\mathcal{J}} C_j\right] \in O(\Delta^{5/2}m).$$

Stochastic jobs before deterministic jobs Here, we condition on the event $X \geq m$. The probability of the event $X < m$ is bounded from above by $\exp\left(\frac{-m}{4}\right)$ by Theorem 2.3. Therefore, $\mathbb{P}[X \geq m] \geq \frac{1}{2}$ for $m \geq 4$. If $X \geq m$, then every machine receives at least one stochastic job before it starts processing the first deterministic job. Thus,

$$\mathbb{E}\left[\sum_{j \in \mathcal{J}_d} C_j \,\middle|\, X \geq m\right] \geq \Delta^{3/2} n_d = \Delta^{11/4}.$$

With the law of total expectation we conclude that

$$\mathbb{E}\left[\sum_{j \in \mathcal{J}} C_j\right] \geq \frac{1}{2}\mathbb{E}\left[\sum_{j \in \mathcal{J}} C_j \,\middle|\, X \geq m\right] \in \Omega(\Delta^{11/4}).$$

Thus, on sufficiently many machines, the index policy $\mathcal{J}_s \prec \mathcal{J}_d$ has total cost greater by a factor $\Omega(\Delta^{1/4})$ than that of policy $\mathcal{J}_d \prec \mathcal{J}_s$.

In summary, we have provided two instances $I_1(\Delta, m)$ and $I_2(\Delta, m)$ which are indistinguishable by any index policy. On the one hand, the policy $\mathcal{J}_d \prec \mathcal{J}_s$ has total expected cost greater by a factor of $\mathcal{O}(\Delta^{1/4})$ than the policy $\mathcal{J}_s \prec \mathcal{J}_d$ for the first instance $I_1(\Delta, m)$. On the other hand, the total expected cost of the policy $\mathcal{J}_s \prec \mathcal{J}_d$ is greater by a factor of $\Omega(\Delta^{1/4})$ than $\mathcal{J}_d \prec \mathcal{J}_s$ on the second instance $I_2(\Delta, m)$. Therefore, the approximation ratio of any index policy is at least $\Omega(\Delta^{1/4})$. □

3.3 Upper Bound for Bernoulli-Type Instances

In this section, we show that taking the number of machines and jobs into account allows for a list scheduling policy that is $\mathcal{O}(m)$-approximate even if the deterministic jobs have different sizes. The stochastic jobs still follow the same distribution. The main result is the following theorem. (In [EFMM19], we prove a similar result restricted to identical deterministic jobs.)

Theorem 3.2. *There exists an $\mathcal{O}(m)$-approximate* List Scheduling *policy for Bernoulli-type instances of* $\mathrm{P}|\,|\mathbb{E}[\sum C_j]$.

For proving this result, we scale the given instance such that $\mathbb{E}[P_j] = 1$ for $j \in \mathcal{J}_s$. That is, we assume without loss of generality that deterministic jobs $j \in \mathcal{J}_d$ have processing time p_j and stochastic jobs $j \in \mathcal{J}_s$ have processing time 0 with probability $1 - \frac{1}{l}$ and l with probability $\frac{1}{l}$, where $l > 0$.

Regarding the total scheduling cost of any policy, we observe the following.

Observation 3.3. *Individually scheduling $\mathcal{J}_d$ (in SPT order) or $\mathcal{J}_s$ on m machines starting at time 0 gives a lower bound on the cost of an optimal policy. We denote these job-set individual scheduling costs by $\sum_{j \in \mathcal{J}_t} \mathbb{E}[C_j^0]$, where $t \in \{s, d\}$. The sum of both also is a lower bound on*

the optimum cost,

$$\sum_{j\in\mathcal{J}} \mathbb{E}[\,C_j^*\,] \geq \sum_{j\in\mathcal{J}_d} \mathbb{E}[C_j^0] + \sum_{j\in\mathcal{J}_s} \mathbb{E}[C_j^0],$$

where C_j^* denotes the (random) completion time of j under a fixed optimal policy.

We prove the result of this section, the existence of an $\mathcal{O}(m)$-approximation, through a careful analysis of the relation between the parameters of a Bernoulli-type instance. In the following, the two policies $\mathcal{J}_s \prec \mathcal{J}_d$ and $\mathcal{J}_d \prec \mathcal{J}_s$ refer to List Scheduling where the deterministic jobs additionally are ordered in Shortest Processing Time order, i.e., in non-decreasing p_j. Clearly, the sorting of jobs as well as following the respective priority order when scheduling the jobs has polynomial time complexity. Hence, we make the following observation.

Observation 3.4. *The policies $\mathcal{J}_s \prec \mathcal{J}_d$ and $\mathcal{J}_d \prec \mathcal{J}_s$ are polynomial-time policies.*

First, we consider instances with few deterministic or few stochastic jobs.

Lemma 3.5. *Let $t, t' \in \{s, d\}$, with $t \neq t'$, refer to the two different job types. The policy $\mathcal{J}_t \prec \mathcal{J}_{t'}$ is a 3-approximate policy for Bernoulli-type instances satisfying $n_{t'} \leq 2m$ with non-uniform deterministic job sizes.*

Proof. By Observation 3.4, the policies $\mathcal{J}_s \prec \mathcal{J}_d$ and $\mathcal{J}_d \prec \mathcal{J}_s$ run in polynomial time.

The cost of $\mathcal{J}_t \prec \mathcal{J}_{t'}$ is at most the cost of scheduling $\mathcal{J}_t$ and the cost of scheduling the ith and the $(m+i)$th job in $\mathcal{J}_{t'}$ on machine i (if these jobs exist), starting after the jobs of $\mathcal{J}_t$ on machine i have completed. Let S_j denote the (random) starting time of job $j \in \mathcal{J}$ under the policy $\mathcal{J}_t \prec \mathcal{J}_{t'}$. By linearity of expectation,

$$\begin{aligned}\sum_{j\in\mathcal{J}} \mathbb{E}[C_j] &= \sum_{j\in\mathcal{J}_t} \mathbb{E}[C_j^0] + \sum_{j\in\mathcal{J}'_t} \mathbb{E}[S_j + P_j] \\ &\leq 3 \sum_{j\in\mathcal{J}_t} \mathbb{E}[C_j^0] + \sum_{j\in\mathcal{J}_{t'}} \mathbb{E}[\,C_j^0\,] \\ &\leq 3 \sum_{j\in\mathcal{J}} \mathbb{E}[C_j^*].\end{aligned}$$

□

Based on this result, we assume $n_s > 2m$ and $n_d > 2m$ for the remainder of this section. Next, we distinguish two cases depending on the number of stochastic jobs relative to the number of deterministic jobs. Both cases rely on a careful analysis of the job-set individual cost for the stochastic jobs. More precisely, we show for a set $\mathcal{J}'_d \subseteq \mathcal{J}_d$ with $|\mathcal{J}'_d| \leq n_s$ that the cost of $\mathcal{J}_s \prec \mathcal{J}'_d$ is bounded by $\mathcal{O}(m) \sum_{j\in\mathcal{J}_s\cup\mathcal{J}_d} \mathbb{E}[\,C_j^*\,]$.

Lemma 3.6. *Let $\mathcal{J}'_d \subseteq \mathcal{J}_d$ with $n_s \geq |\mathcal{J}'_d| > 2m$. Then, $\mathcal{J}_s \prec \mathcal{J}'_d$ is an $\mathcal{O}(m)$-approximation for the job set $\mathcal{J}'_d \cup \mathcal{J}_s$.*

For the proof of this lemma, we use the following technical result that gives some properties of the random variable X counting the number of long stochastic jobs. Recall that X_j is the random variable indicating whether or not P_j is equal to l. That is, $X_j \sim \mathrm{Ber}\left(\frac{1}{l}\right)$, and X can be written as $\sum_{j\in\mathcal{J}_s} X_j$.

We consider arbitrary Bernoulli trials with success probability $q \in (0,1)$. Let $X_j \sim \mathrm{Ber}(q)$ for $j \in [n]$ and let $X = \sum_{j=1}^{n} X_j$. Let Z_i be the random variable denoting the position of the ith success, i.e., the ith variable in $\{X_j : X_j = 1, j \in [n]\}$. The following lemma states some elementary properties of Z_i.

Lemma 3.7. *Let $m \in \mathbb{N}$ with $2m < n$. Let $i \in [\lambda m]$ with $\lambda \in \left[\left\lfloor \frac{n}{m} \right\rfloor\right]$ and let $k \in [n]$. Then,*

(i) $\mathbb{E}[Z_i \mid X = k] = \frac{i}{k+1}(n+1)$,

(ii) $\mathbb{E}[Z_i \mid \lambda m \le X < (\lambda+1)m] \le \frac{i}{\lambda m+1}(n+1)$, *and*

(iii) $\mathbb{E}[n - Z_m \mid m \le X < 2m] \ge \frac{n}{4m}$.

Proof. For $r \in [n]$, the random variable $X^{(r)} := \sum_{j=1}^{r} X_j$ follows a binomial distribution with size parameter r and success probability q as the X_j are independent Bernoulli-distributed random variables with success probability q.

Let us recall that $\mathbb{P}[E \mid F] = \frac{\mathbb{P}[E\cap F]}{\mathbb{P}[F]}$ for two events E and F with $\mathbb{P}[F] > 0$.

Ad (i) For $i, z \in [k]$ with $i \le z$,

$$\{Z_i = z\} = \{X_z = 1\} \cap \{X^{(z-1)} = i-1\},$$

i.e., the event that the ith success happens in trial z is equivalent to observing that the zth trial is a success after having seen $i-1$ successes among the first $z-1$ trials.

Intersecting with the event $\{X = k\}$, we obtain

$$\{X_z = 1\} \cap \{X^{(z-1)} = i-1\} \cap \{X = k\} = \{X_z = 1\} \cap \{X^{(z-1)} = i-1\} \cap \{X - X^{(z)} = k-i\}.$$

Since the underlying Bernoulli trials of the three events on the right side are independent, these events are as well. We conclude

$$\mathbb{P}[Z_i = z \mid X = k] = \frac{\mathbb{P}[X_z = 1] \cdot \mathbb{P}\left[X^{(z-1)} = i-1\right] \cdot \mathbb{P}\left[X - X^{(z)} = k-i\right]}{\mathbb{P}[X = k]} = \frac{\binom{z-1}{i-1} \cdot \binom{n-z}{k-i}}{\binom{n}{k}},$$

where we used that $X^{(z-1)}$ and $X - X^{(z)}$ are binomially distributed with success probability q and size parameter $z-1$ and $n-z$, respectively.

With the convention $\binom{r}{q} = 0$ for $r, q \in \mathbb{N}$ with $q > r$, it follows that

$$\binom{n}{k}\mathbb{E}[\, Z_i \mid X = k\,] = \sum_{z=0}^{n} z\,\mathbb{P}[Z_i = z \mid X = k]\binom{n}{k}$$

$$= i \sum_{z=0}^{n} \binom{z}{i} \binom{n-z}{k-i}$$
$$= i \binom{n+1}{k+1}.$$

The last equality follows from the index shift

$$\sum_{z=0}^{n} \binom{z}{i} \binom{n-z}{k-i} = \sum_{z=1}^{n+1} \binom{z-1}{i} \binom{n+1-z}{k-i}$$

and the following observation: The last line in the above calculation asks in how many ways one can pick $k+1$ successes among $n+1$ trials. We can partition this based on the position of the $(i+1)$st success for a fixed i. The $(i+1)$st success can be positioned between the $(i+1)$st and the $(n-k+i)$th trial. If the $(i+1)$st success is at position z, there have to be i successes among the first $z-1$ trials and, since we want to pick $k+1$ successes, the remaining $n+1-z$ trials have to contain $k-i$ successes. Summing over all positions l of the $(i+1)$st success yields the equality.

Ad (ii) With the Law of Total Expectation (Theorem 2.1), we can use (i) to prove the statement as follows. Conditioning Z_i on the event $\{X = k \mid \lambda m \leq k < (\lambda+1)m\}$ yields

$$\mathbb{E}[\, Z_i \mid \lambda m \leq X < (\lambda+1)m \,] = \sum_{k=\lambda m}^{(\lambda+1)m-1} \mathbb{E}[\, Z_i \mid X = k \,] \mathbb{P}[X = k \mid \lambda m \leq X < (\lambda+1)m].$$

Applying (i), we get that this equals

$$\sum_{k=\lambda m}^{(\lambda+1)m-1} \frac{i}{k+1}(n+1) \mathbb{P}[X = k \mid \lambda m \leq X < (\lambda+1)m].$$

Since $\lambda m + 1$ clearly is a lower bound on the denominator of every summand, this is at most

$$\frac{i}{\lambda m+1}(n+1) \sum_{k=\lambda m}^{(\lambda+1)m-1} \mathbb{P}[X = k \mid \lambda m \leq X < (\lambda+1)m]$$

The Law of Total Expectation, Theorem 2.1, concludes the calculation with

$$\mathbb{E}[\, Z_i \mid \lambda m \leq X < (\lambda+1)m \,] \leq \frac{i}{\lambda m+1}(n+1).$$

Ad (iii) With (i) it follows that

$$\begin{aligned} \mathbb{E}[\, n - Z_m \mid X = m \,] &= n - \frac{m}{m+1}(n+1) \\ &= \frac{nm + n - nm - m}{m+1} \\ &\geq \frac{n}{4m}, \end{aligned}$$

where we used $n > 2m$ for the last inequality. Using again the Law of Total Expectation as in (ii), the statement follows. □

Proof of Lemma 3.6. By Observation 3.4, the policy $\mathcal{J}_s \prec \mathcal{J}'_d$ runs in polynomial time.

We analyze the performance of $\mathcal{J}_s \prec \mathcal{J}'_d$ by conditioning on the number X of long jobs. We index the deterministic jobs in order of their processing times, i.e., $p_1 \leq \cdots \leq p_{|\mathcal{J}'_d|}$.

For the case $0 \leq X < m$, let $k \in \mathbb{N}$ with $0 \leq k < m$. If a realization satisfies $X = k$, then there exists at least one machine that does not schedule stochastic jobs and starts processing deterministic jobs at time 0 in SPT order. Thus,

$$\mathbb{E}\left[\sum_{j \in \mathcal{J}} C_j \,\middle|\, X = k \right] \leq k \cdot l + \sum_{j \in \mathcal{J}'_d} j \cdot p_j.$$

Using a bound by Eastman, Even, and Isaacs [EEI64] on the single-machine scheduling cost in terms of the cost of a schdule on m machines, we have

$$\mathbb{E}\left[\sum_{j \in \mathcal{J}} C_j \,\middle|\, X = k \right] \leq k \cdot l + m \sum_{j \in \mathcal{J}'_d} C^0_j.$$

Since the optimal policy also has to process the k long stochastic jobs,

$$\mathbb{E}\left[\sum_{j \in \mathcal{J}} C^*_j \,\middle|\, X = k \right] \geq k \cdot l + \sum_{j \in \mathcal{J}'_d} C^0_j,$$

where C^*_j denotes the completion time of j under an optimal policy. Thus,

$$\mathbb{E}\left[\sum_{j \in \mathcal{J}} C_j \,\middle|\, X = k \right] \leq m \mathbb{E}\left[\sum_{j \in \mathcal{J}} C^*_j \mid X = k \right].$$

Consider now the case $\lambda m \leq X < (\lambda + 1)m$ for $\lambda \in \left\{1, \ldots, \left\lfloor \frac{n_s}{m} \right\rfloor\right\}$. All stochastic jobs are finished at the latest by time $(\lambda + 1)\, l$. Hence, from time $(\lambda + 1)\, l$ on, all machines process deterministic jobs only. Thus,

$$\sum_{j \in \mathcal{J}} \mathbb{E}[C_j \mid \lambda m \leq X < (\lambda + 1)m] \leq \sum_{j \in \mathcal{J}} \mathbb{E}[C^0_j \mid \lambda m \leq X < (\lambda + 1)m] + (\lambda + 1) l |\mathcal{J}'_d|.$$

As noted in Observation 3.3, the first term is a lower bound on the optimal cost and it remains to bound the second term, i.e., $(\lambda + 1) l |\mathcal{J}'_d|$.

Note that a non-anticipatory policy does not know the positions of the long jobs. Thus, such a policy cannot start any of the stochastic jobs coming after the first $(k \cdot m)$ long ones before time $k \cdot l$ for $1 \leq k \leq \lambda$. Recall that Z_{km} is the (random) position of the $(k \cdot m)$th long job. Hence, $n_s - Z_{km}$ stochastic jobs are delayed by at least $k \cdot l$.

For $\lambda = 1$, Lemma 3.7 (iii) implies that scheduling only $\mathcal{J}_s$ costs at least $l \frac{n_s}{4m}$, i.e.,

$$\sum_{j \in \mathcal{J}_s} \mathbb{E}[C^0_j \mid m \leq X < 2m] \geq l \, \frac{n_s}{4m} \geq \frac{1}{8m} (\lambda + 1) l |\mathcal{J}'_d|.$$

For $2 \leq \lambda \leq \lfloor \frac{n_s}{m} \rfloor$, with Lemma 3.7 (ii) it follows

$$\begin{aligned}\sum_{j \in \mathcal{J}_s} \mathbb{E}[C_j^0 \mid \lambda m \leq X < (\lambda+1)m] &\geq \sum_{k=1}^{\lambda} l\mathbb{E}[\, n_s - Z_{km} \mid \lambda m \leq X < (\lambda+1)m \,] \\ &\geq l n_s \sum_{k=1}^{\lambda} \frac{\lambda m - km}{2\lambda m} \\ &\geq \frac{l\lambda |\mathcal{J}_d'|}{8} \\ &\geq \frac{1}{16}(\lambda+1) l |\mathcal{J}_d'|.\end{aligned}$$

Using again the Law of Total Expectation (Theorem 2.1) and combining the above results,

$$\sum_{j \in \mathcal{J}} \mathbb{E}[C_j] \leq \max\{16, 8m\} \sum_{j \in \mathcal{J}} \mathbb{E}[C_j^*].$$

This concludes the proof. □

If the instance contains less deterministic jobs than stochastic jobs, then Lemma 3.6 immediately implies that $\mathcal{J}_s \prec \mathcal{J}_d$ is an $\mathcal{O}(m)$-approximate policy.

Lemma 3.8. *The policy $\mathcal{J}_s \prec \mathcal{J}_d$ is $\mathcal{O}(m)$-approximate if $2m < n_d \leq n_s$.*

For the case with more deterministic jobs than stochastic jobs, we partition the deterministic jobs into two parts $\mathcal{J}_{d,1}$ and $\mathcal{J}_{d,2}$ and use the List Scheduling order $\mathcal{J}_{d,1} \prec \mathcal{J}_s \prec \mathcal{J}_{d,2}$ where the deterministic jobs are again ordered by non-decreasing processing times. The first set $\mathcal{J}_{d,1}$ contains the deterministic jobs $\{1, \dots, j_d\}$ and the second set $\mathcal{J}_{d,2}$ the jobs $\{j_d + 1, \dots, n_d\}$. Choosing job j_d maximal such that $n_s C_{j_d}^0 \leq \sum_{j \in \mathcal{J}_d} C_j^0$ allows us to simultaneously bound the cost incurred due to scheduling $\mathcal{J}_s$ after $\mathcal{J}_{d,1}$ and due to scheduling $\mathcal{J}_{d,2}$ after $\mathcal{J}_s$. This job exists since $n_s \leq n_d$ and $\sum_{j \in \mathcal{J}_d} C_j^0 \geq n_d C_1^0$. (Recall that the job-set individual cost for the deterministic jobs corresponds to scheduling in SPT order.)

Lemma 3.9. List Scheduling *in the order $\mathcal{J}_{d,1} \prec \mathcal{J}_s \prec \mathcal{J}_{d,2}$ is an $\mathcal{O}(m)$-approximate policy.*

Proof. Since $\sum_{j \in \mathcal{J}_d} C_j^0$ can be computed in polynomial time, job j_d can be determined in polynomial time as well. Hence, the policy $\mathcal{J}_{d,1} \prec \mathcal{J}_s \prec \mathcal{J}_{d,2}$ runs in polynomial time.

The scheduling cost of the policy $\mathcal{J}_{d,1} \prec \mathcal{J}_s \prec \mathcal{J}_{d,2}$ can be naturally split into the three parts corresponding to the respective job sets. Clearly,

$$\sum_{j \in \mathcal{J}_{d,1}} \mathbb{E}[\, C_j \,] \leq \sum_{j \in \mathcal{J}_d} C_j^0.$$

Observe that the policy starts scheduling stochastic jobs no later than $C_{j_d}^0$ by construction.

Hence, the cost incurred by the stochastic jobs can be bounded by

$$\sum_{j\in\mathcal{J}_s}\mathbb{E}[\,C_j\,] \le n_s C^0_{j_d} + \sum_{j\in\mathcal{J}_s}\mathbb{E}[\,C^0_j\,] \le \sum_{j\in\mathcal{J}}\mathbb{E}[\,C^0_j\,].$$

It remains to bound the cost of scheduling $\mathcal{J}_{d,2}$ after the stochastic jobs in $\mathcal{J}_s$. These cost are bounded by List Scheduling at time 0 in the order $\mathcal{J}_s \prec \mathcal{J}_{d,2}$ and adding $|\mathcal{J}_{d,2}|C^0_{j_d}$, i.e.,

$$\sum_{j\in\mathcal{J}_{d,2}}\mathbb{E}[\,C_j\,] \le |\mathcal{J}_{d,2}|C^0_{j_d} + \sum_{j\in\mathcal{J}_{d,2}\cup\mathcal{J}_s}\mathbb{E}[\,C^{\mathrm{LS}}_j\,].$$

The first term on the right side is upper bounded by the set-individual scheduling cost of the deterministic jobs as SPT implies $C^0_j \le C^0_k$ for $j < k$. The second term can be bounded with Lemma 3.6 if $|\mathcal{J}_{d,2}| \le n_s$. As j_d was chosen maximal, it holds that $n_s C^0_{j_d+1} > \sum_{j\in\mathcal{J}_d} C^0_j$. This implies

$$\sum_{j\in\mathcal{J}_{d,2}} C^0_j \ge |\mathcal{J}_{d,2}|C^0_{j_d+1} > \frac{|\mathcal{J}_{d,2}|}{n_s}\sum_{j\in\mathcal{J}_d} C^0_j,$$

where C^0_j, for $j \in \mathcal{J}_{d,2}$, refers to the set-individual cost when scheduling all deterministic jobs. Rearranging yields

$$|\mathcal{J}_{d,2}| \le n_s \frac{\sum_{j\in\mathcal{J}_{d,2}} C^0_j}{\sum_{j\in\mathcal{J}_d} C^0_j} \le n_s.$$

Lemma 3.6 and the above observation on completion times under SPT imply that

$$\sum_{j\in\mathcal{J}_{d,2}}\mathbb{E}[\,C_j\,] \le \sum_{j\in\mathcal{J}_d} C^0_j + \mathcal{O}(m)\sum_{j\in\mathcal{J}_s\cup\mathcal{J}_{d,2}}\mathbb{E}[\,C^*_j\,].$$

Combining the cost individually incurred by the sets $\mathcal{J}_{d,1}, \mathcal{J}_s$, and $\mathcal{J}_{d,2}$, we obtain

$$\sum_{j\in\mathcal{J}}\mathbb{E}[\,C_j\,] \le \mathcal{O}(m)\sum_{j\in\mathcal{J}}\mathbb{E}[\,C^*_j\,].$$

□

We conclude with a policy for scheduling Bernoulli-type instance.

Algorithm 3.1: List scheduling policy for Bernoulli-type instances with non-uniform deterministic jobs

At any time when a machine is idle, select the next job to schedule according to the following priority order:
if $m = 1$ do
    use SEPT
else-if $n_d \le 2m$ do
    use $\mathcal{J}_s \prec \mathcal{J}_d$
else-if $n_s \le 2m$ do

use $\mathcal{J}_d \prec \mathcal{J}_s$
else-if $n_s \geq n_d$ do
use $\mathcal{J}_s \prec \mathcal{J}_d$
else
$j_d \leftarrow \max\{j \in \mathcal{J}_d : n_s C_j^0 \leq \sum_{k \in \mathcal{J}_d} C_k^0\}$
$\mathcal{J}_{d,1} \leftarrow \{1, \dots, j_d\}$
$\mathcal{J}_{d,2} \leftarrow \{j_d + 1, \dots, n_d\}$
use $\mathcal{J}_{d,1} \prec \mathcal{J}_s \prec \mathcal{J}_{d,2}$

Proof of Theorem 3.2. Algorithm 3.1 is a list scheduling policy that selects one out of four index policies, SEPT, $\mathcal{J}_d \prec \mathcal{J}_s$, $\mathcal{J}_s \prec \mathcal{J}_d$, and $\mathcal{J}_{d,1} \prec \mathcal{J}_s \prec \mathcal{J}_{d,2}$, depending on the numbers of jobs and machines. The approximation ratio follows from the fact that SEPT is optimal on a single machine [Rot66] and from Lemmas 3.5, 3.8, and 3.9. Since we can select the correct case in polynomial time and any of these policies runs in polynomial time, Algorithm 3.1 is indeed an $\mathcal{O}(m)$-approximate policy. □

3.4 Further Results on Bernoulli-Type Instances

After having shown the $\mathcal{O}(m)$-approximate policy for Bernoulli-type instances with arbitrary deterministic jobs, we further analyze instances with identical deterministic jobs. We are particularly interested in relationships between relevant parameters that allow us to obtain constant approximate scheduling policies. This interest guides the structure of this section: We dedicate a part to each combination of parameters for which we are able to obtain constant approximation ratios. Interestingly, there remains a gap depending on the expected number of long stochastic jobs. Hence, this section does not provide an $\mathcal{O}(1)$-approximate policy for every Bernoulli-type instance.

3.4.1 Less Stochastic Than Deterministic Jobs

We show that $\mathcal{J}_d \prec \mathcal{J}_s$ is $\mathcal{O}(1)$-approximate if there are less stochastic than deterministic jobs.

Lemma 3.10. *Let $c \in \mathbb{N}$. The policy $\mathcal{J}_d \prec \mathcal{J}_s$ is a $(2c+1)$-approximate policy for Bernoulli-type instances with uniform deterministic job sizes if $n_d > m$ and $n_s \leq c n_d$.*

Proof. When scheduling in order $\mathcal{J}_d \prec \mathcal{J}_s$, machines start processing jobs in $\mathcal{J}_s$ no later than time $\left\lceil \frac{n_d}{m} \right\rceil p \leq 2\frac{n_d}{m} p$ when all jobs in $\mathcal{J}_d$ have completed. Thus, the total cost of scheduling $\mathcal{J}_s$ after $\mathcal{J}_d$ is

$$\sum_{j \in \mathcal{J}_s} \mathbb{E}[C_j^0] + n_s \cdot 2\frac{n_d}{m} p \leq \sum_{j \in \mathcal{J}_s} \mathbb{E}[C_j^0] + 2c \sum_{j \in \mathcal{J}_d} \mathbb{E}[C_j^0] \,.$$

Adding the job-set individual cost of the deterministic jobs $\mathcal{J}_d$ implies the approximation ratio $(2c+1)$ and Observation 3.4 bounds the running time. □

3.4.2 Many Long Stochastic Jobs in Expectation

Recall that X_j denotes the random variable indicating whether or not $P_j = l$ and X counts the number of such stochastic jobs j. Motivated by Lemma 3.7, that analyzes the expected number of stochastic jobs after having observed a certain number of long ones, we consider instances where, in expectation, the number of stochastic long jobs is at least a constant fraction more than the number of machines. We show that $\mathcal{J}_s \prec \mathcal{J}_d$ achieves a constant approximation ratio if additionally $n_d \leq n_s$ holds.

Lemma 3.11. *For Bernoulli-type instances satisfying $n_d \leq n_s$ and $\mathbb{E}[X] \geq (1+\varepsilon)m$, the scheduling policy $\mathcal{J}_s \prec \mathcal{J}_d$ is a $\frac{4+3\varepsilon}{\varepsilon}$-approximate policy.*

Proof. Again, Observation 3.4 bounds the running time of $\mathcal{J}_s \prec \mathcal{J}_d$.

The cost for stochastic jobs incurred by the policy $\mathcal{J}_s \prec \mathcal{J}_d$ is bounded by the set-individual scheduling cost, i.e., by $\sum_{j\in\mathcal{J}_s} \mathbb{E}[C_j^0]$. Hence, it remains to bound the cost for scheduling the deterministic jobs after all stochastic jobs finish. This cost can be bounded from above by $n_d C_{\max}(\mathcal{J}_s) + \sum_{j\in\mathcal{J}_d} C_j^0$, where $C_{\max}(\mathcal{J}_s)$ is a random variable representing the makespan of the stochastic jobs. Since we use List Scheduling, the makespan of $\mathcal{J}_s$ is given by $l\left\lceil \frac{X}{m} \right\rceil$, which is at most $l\left(\frac{X}{m}+1\right)$. Hence, in expectation, the scheduling cost of the deterministic jobs is bounded by

$$n_d\,\mathbb{E}[C_{\max}(\mathcal{J}_s)] \leq l n_d \left(\frac{\mathbb{E}[X]}{m}+1\right) = l n_d + \frac{n_d n_s}{m} \leq \left(1+\frac{1}{1+\varepsilon}\right)\frac{n_s^2}{m},$$

where we used $n_d \leq n_s$ and $\frac{n_s}{l} = \mathbb{E}[X] \geq (1+\varepsilon)m$.

For bounding the term on the right side, we use a valid inequality for *any* scheduling policy Π and *any* subset of jobs $\mathcal{J}' \subseteq \mathcal{J}$ discovered by Möhring, Schulz, and Uetz [MSU99]:

$$\sum_{j\in\mathcal{J}'} \mathbb{E}[P_j]\mathbb{E}\left[C_j^{\Pi}\right] \geq \frac{1}{2m}\left(\sum_{j\in\mathcal{J}'}\mathbb{E}[P_j]\right)^2 + \frac{1}{2}\sum_{j\in\mathcal{J}'}\mathbb{E}[P_j]^2 - \frac{m-1}{2m}\sum_{j\in\mathcal{J}'}\mathbb{V}\mathrm{ar}[P_j].$$

Note that the set-individual cost of the stochastic jobs is given by the cost of List Scheduling in an arbitrary order. Applying this observation and the above bound to $\mathcal{J}' = \mathcal{J}_s$ and using $\mathbb{E}[P_j] = 1$ as well as $\mathbb{V}\mathrm{ar}[P_j] = l-1$, we obtain

$$\begin{aligned}\sum_{j\in\mathcal{J}_s}\mathbb{E}[C_j^0] &\geq \frac{1}{2m}\left(\sum_{j\in\mathcal{J}_s}1\right)^2 + \frac{1}{2}\sum_{j\in\mathcal{J}_s}1^2 - \frac{m-1}{2m}\sum_{j\in\mathcal{J}_s}(l-1)\\ &\geq \frac{n_s^2}{2m} - \frac{n_s l}{2}\\ &\geq \frac{\varepsilon}{2(1+\varepsilon)}\frac{n_s^2}{m},\end{aligned}$$

where we used again that $\frac{n_s}{l} \geq (1+\varepsilon)m$ by assumption. Combining these two bounds yields

$$\sum_{j\in\mathcal{J}} \mathbb{E}[\,C_j\,] \leq \left(1+\frac{1}{1+\varepsilon}\right)\frac{n_s^2}{m} + \sum_{j\in\mathcal{J}_d} C_j^0 + \sum_{j\in\mathcal{J}_s} \mathbb{E}[\,C_j^0\,] \leq \frac{4+3\varepsilon}{\varepsilon} \sum_{j\in\mathcal{J}} \mathbb{E}[\,C_j^*\,].$$

□

3.4.3 Bounded Processing Times of Stochastic Jobs

In this section, we consider Bernoulli-type instances where the processing time of the stochastic jobs is bounded from above by the average load per machine caused by deterministic jobs. We show that, if $P_j \leq 2\frac{n_d p}{m}$ for all jobs and all realizations, then $\mathcal{J}_s \prec \mathcal{J}_d$ achieves a constant approximation ratio.

Lemma 3.12. *For Bernoulli-type instances satisfying $n_d \leq n_s$, $\mathbb{E}[\,X\,] \leq (1+\varepsilon)m$ and $l \leq 2\frac{n_d p}{m}$, the policy $\mathcal{J}_s \prec \mathcal{J}_d$ is a $(9+4\varepsilon)$-approximate policy.*

Proof. As in the proof of Lemma 3.11, the cost for scheduling the deterministic jobs after finishing all stochastic jobs is bounded from above by $n_d C_{\max}(\mathcal{J}_s) + \sum_{j\in\mathcal{J}_d} C_j^0$. In expectation, the first term can be bounded by

$$n_d \mathbb{E}[\,C_{\max}(\mathcal{J}_s)\,] \leq l n_d + \frac{n_d n_s}{m}.$$

Using first $\frac{n_s}{l} \leq (1+\varepsilon)m$ and then $l \leq 2\frac{n_d p}{m}$ yields

$$n_d \mathbb{E}[\,C_{\max}(\mathcal{J}_s)\,] \leq 2(2+\varepsilon)\frac{n_d^2 p}{m}.$$

By a result of Eastman, Even, and Isaacs [EEI64], this term can be bounded by

$$n_d \mathbb{E}[\,C_{\max}(\mathcal{J}_s)\,] \leq 4(2+\varepsilon) \sum_{j\in\mathcal{J}_d} C_j^0.$$

Combining these results with Observation 3.3 on the job-set individual cost and Observation 3.4 on the running time of $\mathcal{J}_s \prec \mathcal{J}_d$ concludes the proof. □

3.4.4 Bounded Makespan of Deterministic Jobs

In this section, we analyze the policy $\mathcal{J}_d \prec \mathcal{J}_s$ for Bernoulli-type instances where the makespan of deterministic jobs is bounded by a constant. We show that this policy achieves a constant approximation ratio.

Lemma 3.13. *If a Bernoulli-type instance satisfies $\frac{n_d p}{m} \leq c$, then $\mathcal{J}_d \prec \mathcal{J}_s$ is a $(c+2)$-approximate policy.*

Proof. For the policy $\mathcal{J}_d \prec \mathcal{J}_s$, the scheduling cost caused by the deterministic jobs coincides with the corresponding job-set individual cost, i.e., with $\sum_{j\in\mathcal{J}_d} C_j^0$. The cost of scheduling the stochastic jobs after the deterministic jobs, is at most $n_s C_{\max}(\mathcal{J}_d) + \sum_{j\in\mathcal{J}_s} \mathbb{E}[\,C_j^0\,]$. As

the deterministic jobs are processed without idle time by List Scheduling, the first term is bounded by

$$n_s C_{\max}(\mathcal{J}_d) = n_s \left\lceil \frac{n_d p}{m} \right\rceil \leq n_s \left(1 + \frac{n_d p}{m}\right).$$

Since $\sum_{j \in \mathcal{J}_s} \mathbb{E}[\, P_j \,] = n_s$ is a valid lower bound on the job-set individual cost for the stochastic jobs and since $\frac{n_d p}{m} \leq c$ by assumption, we have

$$n_s C_{\max}(\mathcal{J}_d) \leq (c+1) \sum_{j \in \mathcal{J}_s} \mathbb{E}[\, C_j^0 \,].$$

Combining these bounds and using Observations 3.3 and 3.4 shows the statement. □

3.4.5 At least $m - 1$ Expected Long Stochastic Jobs

In this part, we consider Bernoulli-type instances where the number of expected long jobs is at least $m-1$, i.e., $\mathbb{E}[\, X \,] \geq m-1$. Depending on the size of deterministic jobs relative to $\mathbb{E}[\, P_j \,]$, we distinguish two cases. If $p \leq \mathbb{E}[\, P_j \,]$, then we show that $\mathcal{J}_d \prec \mathcal{J}_s$ is a constant approximate policy while $\mathcal{J}_s \prec \mathcal{J}_d$ achieves a constant approximation ratio if $p \geq \mathbb{E}[\, P_j \,]$.

A key ingredient to both results of this section is again the following valid inequality by Möhring, Schulz, and Uetz [MSU99] for any scheduling policy Π and any job subset $\mathcal{J}' \subseteq \mathcal{J}$:

$$\sum_{j \in \mathcal{J}'} \mathbb{E}[\, P_j \,] \mathbb{E}\Big[\, C_j^{\Pi} \,\Big] \geq \frac{1}{2m} \left(\sum_{j \in \mathcal{J}'} \mathbb{E}[\, P_j \,] \right)^2 + \frac{1}{2} \sum_{j \in \mathcal{J}'} \mathbb{E}[\, P_j \,]^2 - \frac{m-1}{2m} \sum_{j \in \mathcal{J}'} \mathbb{V}\mathrm{ar}[P_j].$$

Applied to $\mathcal{J}' = \mathcal{J}_s \cup \mathcal{J}_d$, we simplify this to

$$\begin{aligned} \sum_{j \in \mathcal{J}_s} \mathbb{E}\Big[\, C_j^{\Pi} \,\Big] + p \sum_{j \in \mathcal{J}_d} \mathbb{E}\Big[\, C_j^{\Pi} \,\Big] &\geq \frac{1}{2m}\Big(n_s^2 + 2 n_d n_s p + n_d^2 p^2\Big) + \frac{1}{2}\Big(n_s + n_d p^2\Big) - \frac{m-1}{2m} n_s (l-1) \\ &\geq \frac{n_d n_s p}{m}, \end{aligned} \tag{3.1}$$

where we used that $\mathbb{E}[\, X \,] \geq m-1$ implies that $n_s \geq (m-1)(l-1)$.

Lemma 3.14. *For any Bernoulli-type instance satisfying $p \leq 1$, $\frac{n_d p}{m} \geq 1$, and $\mathbb{E}[\, X \,] \geq m-1$, the policy $\mathcal{J}_d \prec \mathcal{J}_s$ is a 3-approximate policy.*

Proof. We bound again the cost of scheduling the stochastic jobs after the deterministic jobs by $n_s C_{\max}(\mathcal{J}_d) + \sum_{j \in \mathcal{J}_s} \mathbb{E}[\, C_j^0 \,]$, where the first term is bounded:

$$n_s C_{\max}(\mathcal{J}_d) \leq n_s \left(\frac{n_d p}{m} + 1 \right) \leq \frac{2 n_s n_d p}{m}.$$

Recall that $\mathbb{E}[\,P_j\,] = 1$ for $j \in \mathcal{J}_s$. Using this and $p \leq 1$, we apply Equation (3.1) to obtain

$$\sum_{j\in\mathcal{J}} \mathbb{E}[\,C_j^*\,] \geq \sum_{j\in\mathcal{J}_s} \mathbb{E}[\,C_j^*\,] + p \sum_{j\in\mathcal{J}_d} \mathbb{E}[\,C_j^*\,] \geq \frac{n_d n_s p}{m}.$$

Therefore, the policy $\mathcal{J}_d \prec \mathcal{J}_s$ achieves an approximation ratio of 3. Further, Observation 3.4 bounds the running time of $\mathcal{J}_d \prec \mathcal{J}_s$. □

Lemma 3.15. *For any Bernoulli-type instance satisfying $p \geq 1$ and $\mathbb{E}[\,X\,] \geq m - 1 \geq 1$, the policy $\mathcal{J}_s \prec \mathcal{J}_d$ is a 4-approximate policy.*

Proof. Observation 3.4 gives the desired polynomial bound on the running time.

The cost for scheduling the deterministic jobs after the stochastic jobs can again be bounded by $n_d l \left\lceil \frac{X}{m} \right\rceil + \sum_{j\in\mathcal{J}_d} C_j^0$ per realization. In expectation, the first term is at most $n_d\left(\frac{n_s}{m} + l\right)$. Using the assumption $\mathbb{E}[\,X\,] \geq m-1$ in the form $l \leq \frac{n_s}{m-1} \leq \frac{2n_s}{m}$, this term is at most $n_d\left(\frac{3n_s}{m}\right)$.

In order to bound this term, we use again Equation (3.1) to obtain

$$p \sum_{j\in\mathcal{J}} \mathbb{E}[\,C_j^*\,] \geq \sum_{j\in\mathcal{J}} \mathbb{E}[\,P_j\,]\mathbb{E}[\,C_j^*\,] \geq \frac{n_d n_s p}{m}.$$

Dividing by p bounds the cost of the scheduling policy $\mathcal{J}_s \prec \mathcal{J}_d$ by $4\sum_{j\in\mathcal{J}} \mathbb{E}[\,C_j^*\,]$. □

3.4.6 Discussion

By combining the previous results, we observe that the remaining cases for Bernoulli-type instances satisfy $n_s > n_d$, $\frac{n_p p}{m} \geq c$ and $\mathbb{E}[\,X\,] < m - 1$. The difficulty with such instances lies in the fact that there is a positive probability that $\mathcal{J}_s \prec \mathcal{J}_d$ blocks all machines and delays all deterministic jobs past time l, the long processing time of the stochastic jobs.

As we have also seen in Section 3.3, if the realized number of long jobs exceeds the number of machines by at least a constant fraction, then the set-individual cost for scheduling the stochastic jobs is sufficient for bounding the cost of the deterministic ones. However, if we only have constantly many more realized long jobs than machines, then the set-individual cost only allows for an $\mathcal{O}(m)$-approximate policy but does not yield a constant performance guarantee.

Interestingly, known results about stochastic scheduling policies can be used to obtain $\mathcal{O}(1)$-approximate policies if the expected number of long jobs is at least $m-1$ which in turn implies a better lower bound on any scheduling policy. If an instance has at most $m-2$ long jobs in expectation, then the large processing time of stochastic jobs (and hence the additional cost for delaying the deterministic jobs by l) is too large to be balanced by the probability that many stochastic processing times turn out to be long. Hence, the cost of $\mathcal{J}_s \prec \mathcal{J}_d$ cannot be bounded with the lower bounds on an optimal policy that are currently in use. Similarly, with the lower bounds currently known, the sheer number of stochastic jobs prevents $\mathcal{J}_d \prec \mathcal{J}_s$ from being $\mathcal{O}(1)$-approximate. We also tried to analyze "mixed" policies where we schedule stochastic jobs

until a constant fraction of machines is blocked with long jobs before completely switching to deterministic jobs. The problem here is that we force ourselves to delay a considerable fraction of stochastic jobs to be scheduled *after* all deterministic jobs while an optimal policy might be lucky in the same realization and schedule all of these jobs at time zero.

It remains an interesting open question whether Bernoulli-type instances allow for $\mathcal{O}(1)$-approximate policies. We believe that better lower bounds on the cost of the optimal scheduling policy are necessary to improve upon our $\mathcal{O}(m)$-approximate policy. Except for the bound by Möhring, Schulz, and Uetz [MSU99], we are not capable of exploiting the fact that any policy has to schedule both types of jobs and hence incurs some "mixed" cost, i.e., the cost for scheduling a subset of the jobs of one type before the other type (and thus delaying the second) or assigning a particular type of jobs only to a subset of machines.

3.5 Concluding remarks

In this chapter, we rule out distribution-independent approximation factors for minsum scheduling for simple index policies, including List Scheduling in SEPT, LEPT, and LVF order. This strong lower bound holds even for Bernoulli-type instances. It may surprise that such simple, yet stochastic, instances already seem to capture the inherent difficulties of adaptive stochastic scheduling. We believe that understanding the seemingly most simple Bernoulli-type instances is a key for making progress on approximate policies for stochastic scheduling problems. The general importance of high-variance jobs has also been observed in earlier work [MSU99, MUV06, Sch08, IMP15, GMUX17].

For Bernoulli-type instances with arbitrary deterministic jobs, we also give an $\mathcal{O}(m)$-approximate list scheduling policy. The key ingredient to this analysis is the improved lower bound on the optimal cost due to exploiting the properties of the underlying probability distributions. It would be a major improvement to generalize this lower bound to arbitrary probability distributions. Generally, it is a common understanding that improving upon lower bounds is fundamental for designing $\mathcal{O}(1)$-approximate scheduling policies.

The setting with a fixed number m of machines is of particular interest. While the special case $m = 1$ is solved optimally by SEPT [Rot66], even the problem on $m = 2$ machines is open. For Bernoulli-type instances, the index policy we give in this note is, in fact, a constant-factor approximation. Any generalization would be of interest. Notice that our lower bound for arbitrary index policies as well as earlier lower bounds on SEPT [CFMM14, IMP15] rely on a large number of machines. Thus, even SEPT or some other simple index policy might give a constant factor approximation for constant or bounded m.

For general instances, our lower bound for index policies suggests that future research on more sophisticated scheduling policies is necessary for $\mathcal{O}(1)$-approximate policies.

4

Online Load Balancing with Reassignment

We investigate an online variant of load balancing with restricted assignment. In the offline setting, there are n jobs given which need to be processed by m machines with the goal to minimize the maximum machine load. Each job j has a processing time p_j and can only be processed by a subset of the machines. In the online variant of this model, the jobs are only revealed incrementally and have to be immediately assigned to a machine before the next job is revealed.

Since there exist strong lower bounds even for the special case of $p_j = 1$ for all j, we allow our online algorithm to reassign a job j at a cost of $c_j \geq 0$. This model contains two online models as special cases: The model with unit reassignment cost is referred to as recourse model while migration refers to $c_j = p_j$. In this chapter, we generalize a result by Gupta, Kumar, and Stein [GKS14] on online load balancing with recourse to the setting with arbitrary cost. That is, for unit processing times, we maintain a constant competitive assignment with reassignment cost linear in $\sum_j c_j$. For arbitrary processing times, we give an $\mathcal{O}(\log \log mn)$-competitive algorithm with reassignment cost $\mathcal{O}(1) \sum_j c_j$.

Bibliographic Remark: This chapter is based on unpublished, joint work with S. Berndt and N. Megow.

Table of Contents

4.1 Introduction

We analyze an algorithm for an online scheduling problem. A set $\mathcal{J}$ of n jobs has to be assigned to m machines to minimize the maximum load $C_{\max}$. We focus on the case of *restricted assignment* where each job is characterized by a processing time $p_j \in \mathbb{N}$ and a set of machines it is allowed to be processed by. The online model we consider is based on the *online-list* model, where jobs are revealed one by one and any online algorithm has to irrevocably assign the job to one of its machines before the next job is revealed. That is, the jobs are revealed in the order $1, \dots, n$, and upon arrival of job j, the scheduler learns the processing time p_j and the set of machines that can process j. The algorithm has to assign j to one of its machines before job $j+1$ is revealed.

Restricted assignment is a special case of scheduling on unrelated machines where each job j has a processing time $p_{i,j}$ that depends on the machine i the job is assigned to. Restricted assignment can be modeled by requiring $p_{i,j} \in \{p_j, \infty\}$ for each job j and each machine i. We note that all known lower bounds on the competitive ratio of online algorithms for load balancing on unrelated machines already hold for restricted assignment.

Azar, Naor, and Rom [ANR92] give a lower bound of $\Omega(\log n)$ for any online algorithm, even if $p_{i,j} \in \{1, \infty\}$. In recent years, several models have been developed to circumvent such lower bounds by either giving the online algorithm more power or decreasing the knowledge or power of the adversary. In this chapter, we choose the former model where the online algorithm is allowed to revoke assignment decisions at a certain cost. That is, upon arrival of a new job, previously revealed and assigned jobs might be reassigned at a job-dependent cost. Of course, if one did not impose any bound on these reassignment cost, then the algorithm could simulate the current offline optimum. Therefore, we assume that each job j has a non-negative *assignment cost* c_j that any scheduler has to pay when it (re)assigns j to a particular machine. In particular, the assignment cost of an offline optimum is given by the sum of the assignment costs of the current set of jobs.

As described in Chapter 2, we use competitive analysis to evaluate the performance of an online algorithm. That is, an online algorithm is *c-competitive* if, for each instance and after each arrival of a new job, its makespan is bounded by $c\,C^*_{\max}$, where $C^*_{\max}$ is the minimal makespan of any feasible schedule for the current job set. Further, we say that an online algorithm has a *reassignment factor* of β if its *amortized* reassignment cost over the first k rounds is bounded by $\beta \sum_{j=1}^{k} c_j$ for each $k \in [n]$. The aim is to design a c-competitive online algorithm with bounded reassignment factor.

Migration and recourse The model with reassignment cost generalizes both the recourse model — by setting $c_j = 1$ — and the migration model with $c_j = p_j$. We refer to these special reassignment factors by recourse and migration factor, respectively. We note that in these two special cases the first assignment usually does not incur any cost. Both models have been

analyzed from an amortized as well as from a worst-case point of view. In the latter, the reassignment cost in round k is required to be bounded by βc_k. Clearly, any worst-case bound translates to a bound in the amortized setting while the reverse is not necessarily true.

Westbrook [Wes00] is the first to consider online scheduling with reassignments. He considers the case where jobs may arrive and depart. Here, the optimal makespan may decrease over time. Therefore, he designs algorithms that are c-competitive against the *current* optimal load. He gives constant competitive algorithms with constant migration factor and constant recourse factor, respectively, for identical as well as related machines. For arbitrary reassignment costs, the algorithm is $\mathcal{O}\Big(\log_\delta \frac{\max_j\{c_j/p_j\}}{\min_j\{c_j/p_j\}}\Big)$-competitive with reassignment factor $\mathcal{O}(\delta)$ for some parameter δ with $1 \leq \delta \leq \frac{\max_j\{c_j/p_j\}}{\min_j\{c_j/p_j\}}$. Andrews, Goemans, and Zhang [AGZ99] improve upon these results giving algorithms that are constant competitive against the current optimal load with constant reassignment factor for identical and related machines.

Even for load balancing on identical machines, there is a lower bound of $\sqrt{3} \approx 1.88$ on the competitive ratio of online algorithms by Rudin and Chandrasekaran [IC03] while the best known algorithm achieves a competitive ratio of 1.92 and is due to Albers [Alb99]. Sanders, Sivadasan, and Skutella [SSS09] improve upon this lower bound when using migration. More precisely, they obtain a $\frac{3}{2}$-competitive algorithm with worst-case migration factor $\frac{4}{3}$. Moreover, they design a family of $(1+\varepsilon)$-competitive algorithms with worst-case migration factor $\beta(\varepsilon)$ allowing for trade-off between the quality of a solution and its migration cost. In the online setting, they call such a family of algorithms *robust PTAS*. Also for identical parallel machines, Skutella and Verschae [SV16] develop a robust PTAS for two problems, minimizing the maximum load and maximizing the minimum load on any machine, with an amortized bound on the migration factor. When jobs can be preempted, Epstein and Levin [EL14] give a 1-competitive, i.e., optimal, algorithm with worst-case migration factor $1 - \frac{1}{m}$.

Awerbuch et al. [AAPW01] investigate (among other problems) load balancing on unrelated machines and give an $\mathcal{O}(\log m)$-competitive algorithm reassigning each job at most $\mathcal{O}(\log m)$ times. For the special case where $p_{i,j} \in \{1, \infty\}$ for each job j and each machine i, their algorithm is 16-competitive using $\mathcal{O}(\log m)$ recourse if the optimal makespan is at least $\Omega(\log m)$.

Gupta, Kumar, and Stein [GKS14] give an online algorithm for the general restricted assignment problem that is $\mathcal{O}(\log\log mn)$-competitive with constant recourse. For the special case of restricted assignment with unit-size jobs, they give a $\mathcal{O}(1)$-competitive algorithm with constant recourse. Further, they consider an online flow problem with a single source where sinks arrive online that want to receive one unit of flow from the source. If there is a feasible offline solution with cost C^*, then the algorithm violates the capacities by a factor at most $(2+\varepsilon)$ with rerouting cost at most $\Big(1+\frac{2}{\varepsilon}\Big)C^*$ for $\varepsilon > 0$. The rerouting cost are defined as follows: If the flow on an arc is increased or decreased, then an arc-dependent cost has to be paid per unit.

For restricted assignment with unit-size jobs, Bernstein et al. [BKP$^+$17] give an 8-competitive online algorithm with constant recourse that simultaneously achieves the competitive ratio for

every ℓ_p-norm for $p \in [1, \infty]$. That is, if $l = (l_1, \dots, l_m)$ is the load vector of a given job-to-machine assignment, then the ℓ_p-norm of l is defined by $\sqrt[p]{\sum_{i=1}^{m} l_i^p}$ for $p < \infty$ and ℓ_∞ is $\max_i l_i$. They achieve this by carefully following a particular optimal assignment with machine loads $(l_1^*, \dots, l_m^*)$ such that $l_i \leq 8l_i^*$ after each arrival.

Further related work Azar, Naor, and Rom [ANR92] give a strong lower bound of $\Omega(\log m)$ on the competitive ratio of *any* online algorithm for the restricted assignment problem, even if $p_{i,j} \in \{1, \infty\}$. Since in their example $n = m$, this additionally gives a lower bound of $\Omega(\log n)$. They also give an online algorithm matching this lower bound for the general load balancing problem with restricted assignment. For randomized algorithms, they show that the exact competitive ratio is in $[\ln m, \ln m + 1]$, where $\ln m$ denotes the natural logarithm of m for $m > 0$.

If jobs may arrive and depart, Azar, Broder, and Karlin [ABK94] give a lower bound of $\Omega(\sqrt{m})$ and, since $n = \Theta(m)$ in their lower bound example, simultaneously of $\Omega(\sqrt{n})$. Azar et al. [AKP+97] give an algorithm with matching competitive ratio $\mathcal{O}(\sqrt{m})$.

Recourse and migration in online optimization has been studied for a variety of additional problems; among them matching problems [GKS14, BKP+17, BHR19], connectivity problems, such as Minimum Spanning Tree and Traveling Salesperson [MSVW16] as well as Steiner Tree [GGK16, IW91], and packing problems [EL09, EL13, BJK20, JK19]. Online optimization with reassignment cost has been considered for load balancing problems [Wes00, AGZ99] and for Bin Packing [FFG+18].

Our contribution We generalize the result by Gupta, Kumar, and Stein [GKS14] on online load balancing with recourse to the setting where job reassignments incur job-dependent costs. We are able to match their competitive ratio of $\mathcal{O}(\log \log mn)$ (up to constants) with reassignment cost $\mathcal{O}(1) \sum_{j=1}^{n} c_j$. We note that our result also implies a competitive ratio of $\mathcal{O}(\log \log mn)$ with constant migration factor for online load balancing with migration.

4.2 Online Flows with Rerouting

Our results rely on and are inspired by the online flow algorithm with rerouting designed by Gupta, Kumar, and Stein [GKS14]. Hence, we describe their algorithm in this section.

We consider the following online flow problem. We are given a directed graph $G = (V, A)$ with vertices V and arcs A. Each arc $a \in A$ has a capacity $u_a \in \mathbb{Z}_+$ and a cost $c_a \geq 0$. Moreover, there is a source vertex $s \in V$. In round t, vertex $v_t \in V$ is specified as sink and the task is to (unsplittably) send one unit of flow from s to v_t, in addition to the unit flows already being routed from the source to the vertices $v_1, \dots, v_{t-1}$, without violating the arc capacities u_a. Throughout this chapter we assume that the underlying offline problem admits a feasible solution while an online algorithm may violate some capacity constraints.

Easy examples show that, in order to satisfy all the demands specified by the various sinks, any deterministic online algorithm has to violate the arc capacities to some extent. Then, for determining the quality of an algorithm, we are interested in two properties: (i) the minimal factor by which any arc capacity is violated and (ii) the total cost of the flow. In round t, that is *after* satisfying the demand of vertices $v_1, \dots, v_t$, let $(f_a(t))_{a \in A} \in \mathbb{N}^A$ denote the flow found by the online algorithm. We say that the algorithm is *c-competitive* if $f_a(t) \leq cu_a$ holds for each arc a and each round t. Although this notion of competitiveness is orthogonal to the classical use of describing the ratio between the cost of the optimum and the cost of the algorithm, it allows for an easier description when we ultimately talk about load balancing with restricted assignment.

We observe that this problem generalizes load balancing with restricted assignment and unit-size jobs in the following way. In the offline problem, we create for each machine and for each job one vertex and add one vertex s as source. Given the optimal makespan $C^*_{\max}$, the source connects to each machine-vertex i by an arc with capacity $u_{s,i} = C^*_{\max}$ and cost $c_{s,i} = 0$. Further, between each machine-vertex i and each job-vertex j, we draw an arc (i, j) with capacity 1 and cost c_j if and only if j can be scheduled by machine i, i.e., if $p_{i,j} = 1$. By specifying each job-vertex as sink with unit demand, we obtain an instance of the offline version of the above introduced flow problem. The online flow problem assumes that the graph is known upfront while online load balancing is characterized by having the jobs, i.e., in the reduction the job-vertices, revealed one by one. We emphasize that the graph we created has a very special structure. Before a job-vertex is specified as a sink, sending flow along its incident arcs violates the flow conservation at this vertex since all incident arcs are entering this node. Thus, any algorithm that always maintains a feasible solution to the flow problem will not use any of these arcs. The shortest-augmenting-path algorithm designed in [GKS14] satisfies this condition.

The just developed reduction implies that the lower bound of $\Omega(\log m)$ on the competitive ratio for any online algorithm without reassignment for load balancing with restricted assignment also holds for the online flow problem using the above definition of competitiveness for this problem. To beat this strict lower bound, we allow the online algorithm to reroute flow at a certain cost. More precisely, every time the flow sent along an arc a is decreased or increased by one unit, the cost c_a has to be paid. Let C^* be the cost of an optimal solution after the first t rounds. We aim at developing algorithms that violate the arc capacities by at most a constant factor and simultaneously reroute flow at a cost bounded by $\mathcal{O}(C^*)$.

To this end, we have a closer look at the shortest path algorithm developed by Gupta, Kumar, and Stein [GKS14]. Let f be the flow in graph G after round t. We define the residual network G_t on the vertex set V as follows: For every arc $a \in A$ let $\bar{a}$ be its backward arc, i.e., $a = (v, w)$ and $\bar{a} = (w, v)$. Set $u^t_a = cu_a - f_a$ and $u^t_{\bar{a}} = f_a$, where c is the competitive ratio we are aiming for. Moreover, let $c^t_a = c^t_{\bar{a}} = c_a$. That is, in contrast to the classical shortest-augmenting-path algorithm, the backward arc of every arc with positive flow has cost

identical to its forward arc. If vertex v_t is specified as sink in round t, then use a shortest path algorithm to find P, a shortest path from s to v_t in the residual network G_t. We augment the flow f along P by one unit, i.e., if $a \in P$, then the flow along a is increased by one unit, while $\bar{a} \in P$ implies that f_a is decreased by one unit.

Gupta, Kumar, and Stein show that this algorithm maintains a $(2+\varepsilon)$-competitive flow while the cost of rerouting the flow is at most $\left(1+\frac{2}{\varepsilon}\right)$ times the cost of an offline optimum. We restate their main result on maintaining flows online. For the proof we refer to [GKS14].

Theorem 4.1 (Theorem 6.1 in [GKS14]). *If there is a feasible solution f^* to the flow instance G with source s and sinks $v_1, \ldots, v_t$ of cost C^*, the total cost of augmentations performed by the adapted shortest-augmenting-path algorithm on instance G is at most $\left(1+\frac{2}{\varepsilon}\right)C^*$. The capacities on the arcs are violated by at most a factor of $(2+\varepsilon)$.*

4.3 Online Load Balancing with Reassignment

In this section, we prove the main result of this chapter, namely, the existence of a randomized $\mathcal{O}(\log\log mn)$-competitive online algorithm for load balancing with restricted assignment, whose reassignment cost is bounded by $\mathcal{O}(1)\sum_{j=1}^{n} c_j$. We start by explaining a result by [GKS14] for the special case of unit-size jobs as an immediate corollary of Theorem 4.1. Then, we proceed similarly to the proof of Theorem 8.1 in [GKS14]: We partition the set of jobs according to their processing times into big and small jobs with the classification being relative to the current guess of the makespan. For small jobs, we use the algorithm developed for unit-size jobs to obtain a fractional assignment that will then guide the assignment probabilities of our randomized algorithm. Big jobs are further classified into groups of roughly equal processing time such that the algorithm for unit-size jobs can explicitly handle their assignment. Since we treat each of the $\mathcal{O}(\log\log mn)$ classes of big jobs separately, the loss in the competitive ratio compared to the online flow problem is immediate.

4.3.1 Unit-Size Jobs

We start by giving an intuition on how we will use the result on online flows for online load balancing with restricted assignment. Consider again the special case with $p_{i,j} \in \{1, \infty\}$ for each job j and each machine i. As explained above, this problem can directly be translated to the online flow problem assuming that $C^*_{\max}$, the optimal makespan, is known in advance. This assumption is not a restriction as we can employ a standard guess-and-double framework at the cost of losing an additional factor of 2 in the competitive ratio. Specifically, we start by guessing $C^*_{\max} = 1$, i.e., we assign the arcs (s,i) for $i \in [m]$ a capacity of $(2+\varepsilon)$, where $\varepsilon > 0$ is the parameter that describes the trade off between competitive ratio and reassignment cost in Theorem 4.1. That is, our algorithm will be $2(2+\varepsilon)$-competitive with reassignment cost at most $\left(1+\frac{2}{\varepsilon}\right)$. In general, let *round* t refer to the point in time when job j_t is revealed.

In general, if $C^*_{\max}$ is the guess of the optimal makespan in *round* t, then the arcs (s, i) for $i \in [m]$ have capacity $(2+\varepsilon)C^*_{\max}$. If the shortest augmenting path algorithm does not find a feasible flow in this network, then Theorem 4.1 implies that the true optimum is strictly greater than $C^*_{\max}$. Hence, we double $C^*_{\max}$ and rerun the shortest augmenting path algorithm on the residual network G_t with the updated capacities $u_{s,i} = (2+\varepsilon)C^*_{\max}$.

As the failure of the shortest augmenting path algorithm before doubling gives a lower bound on the optimal makespan, we obtain the following corollary; see also Section 7 in [GKS14].

Corollary 4.2. *Let $0 < \varepsilon \leq 1$. If there is a feasible solution with makespan $C^*_{\max}$ and assignment cost C^* to the (offline) load balancing problem with restricted assignment and unit-size jobs, then the shortest augmenting path algorithm combined with a guess-and-double framework maintains a schedule with makespan at most $2(2+\varepsilon)C^*_{\max}$ and reassignment cost at most $\left(1+\frac{2}{\varepsilon}\right)C^*$.*

We note that this result may overestimate the actual reassignment cost due to the following observation: In the online flow problem, increasing or decreasing the flow along an arc a by one unit costs c_a. When balancing load online with reassignment, the reassignment of job j costs c_j. However, the reduction we use implies that reassigning one unit-size job j from machine i to machine i' is equivalent to decreasing the flow along the arc (i, j) by one unit while simultaneously increasing the flow along the arc (i', j) by one unit. This implies that the cost for rerouting the unit-flow associated with job j is $2c_j$.

4.3.2 Small Jobs

Our algorithm classifies jobs as *big* and *small* depending on the current guess of the optimal makespan and the total number of jobs. Let us assume that we know n, the number of jobs, and $C^*_{\max}$, the optimal makespan. We justify this assumption when designing the algorithm for the complete instance. Let $\gamma = \log mn$. We say a job j is big if $p_j \geq \frac{C^*_{\max}}{\gamma}$, and otherwise, the job is small. Our algorithm treats these jobs differently, and we start by only considering the small jobs, $\mathcal{J}_S$, of the instance. We prove the following result.

Theorem 4.3. *There is a randomized online algorithm maintaining an assignment of the small jobs $\mathcal{J}_S$ with expected makespan at most $\mathcal{O}(1)C^*_{\max}$ while incurring an expected reassignment cost at most $\mathcal{O}(1)\sum_{j\in\mathcal{J}_S} c_j$.*

For simplicity, we assume that the set $\mathcal{J}_S$ of small jobs is indexed in the order of the arrival of jobs, i.e., $\mathcal{J}_S = \{1, \ldots, n_S\}$, where $n_S = |\mathcal{J}_S|$. For scheduling the small jobs, we first consider a fractional assignment of the jobs to machines in each time step. We interpret this fractional assignment as a probability distribution of the jobs over the machines and would like to obtain an integral assignment by applying classical rounding schemes. However, as we aim at designing an online algorithm with bounded reassignment cost, we cannot round the

solution in round t independently of the solution after round $t-1$ while hoping to control the total reassignment cost. Thus, we follow the careful rounding scheme developed by [GKS14].

Formally, for job j with processing time p_j and assignment cost c_j, we generate p_j unit-size jobs with reassignment cost $\frac{c_j}{p_j}$ and consider them as an input to online load balancing with unit-size jobs as solved in Section 4.3.1. The set of machines that are able to process a unit-size job associated with j is identical to the set of feasible machines for job j. We interpret the assignment of the associated unit-size jobs as fractional assignment of the original job.

Consider round t, i.e., the assignment after job t has arrived and was fractionally assigned by the algorithm in p_t steps, one part per step. We are only interested in the final assignment (of all unit-size jobs) and discard the intermediate assignments while job t was only partially assigned. Let $x_{i,j}(t)$ be the number of unit-size jobs of job j that are assigned to machine i at time t. Then, the total load on machine i at time t is given by $l_i(t) = \sum_{j=1}^{t} x_{i,j}(t)$. Consider a machine i with $x_{i,j}(t) = x_{i,j}(t-1)$. Then, no unit-size job is moved from or to machine i. Hence, the reassignment cost for such a machine is equal to zero. For machine i with $x_{i,j}(t-1) > x_{i,j}(t)$, exactly $x_{i,j}(t-1) - x_{i,j}(t)$ unit-size jobs are moved from machine i to machines i' with $x_{i',j}(t-1) < x_{i',j}(t)$. By definition, reassigning one unit-size job associated with j has actual cost $\frac{c_j}{p_j}$. However, as observed in Section 4.3.1, the transformation to the online flow problem implies that reassigning one unit-size job from i to i' costs us $2\frac{c_j}{p_j}$ as it involves decreasing the flow on the edge between j and i and increasing the flow on the edge between j and i'. Hence, the assignment cost incurred due to the arrival of job t is given by

$$c(t) := \sum_{i=1}^{m} \sum_{j=1}^{t} \frac{c_j}{p_j} \Big| x_{i,j}(t-1) - x_{i,j}(t) \Big|. \tag{4.1}$$

If there is a schedule with makespan $C^*_{\max}$, the algorithm maintains a fractional schedule with makespan at most $6C^*_{\max}$ and reassignment cost at most $\sum_{s=1}^{t} c(s) \leq 3 \sum_{j=1}^{t} c_j$ by setting $\varepsilon = 1$ in Corollary 4.2.

Since we are interested in an assignment of the original jobs j, we need to translate the fractional assignment $(x_{i,j}(t))_{i,j}$ at time t to an integral assignment without significantly increasing the reassignment cost. A standard approach is to interpret the variables $\left(\frac{x_{i,j}(t)}{p_j}\right)_{i=1}^{m}$ as a probability distribution over the possible assignments of job j to the machines. In other words, if $X_j(t) \in [m]$ is the random variable dictating the assignment of j, then $\mathbb{P}[X_j(t) = i] = \frac{x_{i,j}(t)}{p_j}$. Since the unit-size jobs associated with j have the same set of feasible machines, $x_{i,j}(t) = 0$ if $p_{i,j} = \infty$. Hence, the assignment given by $X_j(t)$ for $1 \leq j \leq t$ is feasible.

However, simply drawing the random variables $X_j(t)$ according to the distribution given by $\left(\frac{x_{i,j}(t)}{p_j}\right)_{i=1}^{m}$ does not allow us to bound the reassignment cost of the actual jobs in terms of the bound $c(t)$ defined in Equation (4.1). Therefore, we use the rounding approach developed by [GKS14] that takes the realization of $X_j(t-1)$, i.e., the assignment of j in round $t-1$, into account when drawing the new assignment $X_j(t)$. In round t, the newly arrived job t is always

assigned according to the probabilities $\left(\frac{x_{i,t}(t)}{p_t}\right)_{i=1}^{m}$ since there is no previous assignment that needs to be taken into account.

Fix a small job $j \in \mathcal{J}_S$ with $j < t$. We construct the following complete bipartite directed graph $G(t)$ with vertex set $V(t-1) \cup V(t)$ and arc set $V(t-1) \times V(t)$, denoted by $A(t)$. The two vertex sets $V(t-1)$ and $V(t)$ contain one vertex for each machine, i.e., $V(s) = \{i(s) : i \in [m]\}$ for $s \in \{t-1, t\}$. An arc $a = (i(t-1), i'(t))$ has cost $c_a = 0$ if $i = i'$. Otherwise, the cost for arc $a \in A$ equals the reassignment cost of one of j's unit sized jobs, i.e., $c_a := \frac{c_j}{p_j}$. Each vertex $i(t-1)$ is a source with demand $d_{i(t-1)} = -x_{i,j}(t-1)$, while each vertex $i(t)$ is a sink with demand $d_{i(t)} = x_{i,j}(t)$. Since $\sum_i x_{i,j}(t-1) = p_j = \sum_i x_{i,j}(t)$, we can solve the min-cost transportation problem for the p_j units of flow from $V(t-1)$ to $V(t)$; for details please refer to the book on network flows [AMO93]. Consider now the integral assignment $X_j(t-1) = i$ of j at time t. Then, pick one of the $x_{i,j}(t-1)$ units placed at i uniformly at random independently of other jobs $j' \neq j$. Suppose this unit is sent to node $i'(t)$ by the solution to the transportation problem. Set $X_j(t) = i'$. The following lemma gives some useful properties of the random variables that enable us to bound the reassignment cost of the integral assignment. As these properties are only mentioned but not proven in [GKS14], we provide a full proof here.

Lemma 4.4. *The random variables $X_j(t)$ for $1 \leq j \leq t \leq n_S$ satisfy the following properties:*

(i) $X_j(t)$ *and* $X_{j'}(t)$ *are independent for* $j \neq j'$,

(ii) $\mathbb{P}[X_j(t) = i] = \frac{x_{i,j}(t)}{p_j}$, *and*

(iii) $\mathbb{P}[X_j(t-1) \neq X_j(t)] = \sum_{\substack{i \in [m]: \\ \mathbb{P}[X_j(t-1)=i]>0}} \frac{1}{p_j}(x_{i,j}(t-1) - x_{i,j}(t))^+$, *where* $x^+ = \max\{x, 0\}$.

Proof. We fix a time t.

Ad (i) Solving the transportation problem independently for each job implies Property (i).

Ad (ii) We prove this by induction on round t. Consider $j = 1$, the first small job that arrived. Clearly, $\mathbb{P}[X_1(1) = i] = \frac{x_{i1}(1)}{p_1}$ by definition. Suppose now that (ii) holds for all jobs $1 \leq j \leq t-1$ in round $t-1$. Consider the fractional assignment $(x_{i,j}(t))_{i,j}$ after job t arrived. Let $f_{i,i'}$ denote the flow from machine vertex $i(t-1)$ to vertex $i'(t)$ as given by the optimal solution to the min-cost transportation problem. If $X_j(t-1) = i$, then the probability that $X_j(t) = i'$ is $\frac{f_{i,i'}}{x_{i,j}(t-1)}$. By the Law of Total Expectation (Theorem 2.1) and by the induction hypothesis,

$$\begin{aligned}
\mathbb{P}\Big[X_j(t) = i'\Big] &= \sum_{\substack{i \in [m]: \\ \mathbb{P}[X_j(t-1)=i]>0}} \mathbb{P}\Big[X_j(t) = i' \,\Big|\, X_j(t-1) = i\Big]\mathbb{P}\Big[X_j(t-1) = i\Big] \\
&= \sum_{\substack{i \in [m]: \\ \mathbb{P}[X_j(t-1)=i]>0}} \frac{f_{i,i'}}{x_{i,j}(t-1)} \frac{x_{i,j}(t-1)}{p_j} \\
&= \frac{x_{i',j}(t)}{p_j},
\end{aligned}$$

where the last equality follows from $f_{ii'}$ being a feasible solution to the transportation problem.

Ad (iii) Recall that $c_{i(t-1),i(t)} = 0$. For a machine i with $x_{i,j}(t-1) > x_{i,j}(t)$, the optimal solution to the transportation problem sends $x_{i,j}(t-1) - x_{i,j}(t)$ unit jobs to other machines. Thus, $\mathbb{P}\left[X_j(t) \neq X_j(t-1) \,|\, X_j(t-1) = i\right] = \frac{x_{i,j}(t-1)-x_{i,j}(t)}{x_{i,j}(t-1)}$. For i with $x_{i,j}(t-1) \leq x_{i,j}(t)$ the optimal solution to the transportation problem sends $x_{i,j}(t-1)$ unit jobs from $i(t-1)$ to $i(t)$. Thus, $\mathbb{P}\Big[X_j(t) \neq X_j(t-1) \,|\, X_j(t-1) = i\Big] = 0$. Therefore,

$$\begin{aligned}
\mathbb{P}\Big[X_j(t) \neq X_j(t-1)\Big] &= \sum_{\substack{i\in[m]:\\ \mathbb{P}[X_j(t-1)=i]>0}} \mathbb{P}\Big[X_j(t) \neq X_j(t-1) \,|\, X_j(t-1) = i\Big]\mathbb{P}\Big[X_j(t-1) = i\Big] \\
&= \sum_{\substack{i\in[m]:\\ \mathbb{P}[X_j(t-1)=i]>0}} \frac{\Big(x_{i,j}(t-1) - x_{i,j}(t)\Big)^+}{x_{i,j}(t-1)} \frac{x_{i,j}(t-1)}{p_j} \\
&= \sum_{\substack{i\in[m]:\\ \mathbb{P}[X_j(t-1)=i]>0}} \frac{\Big(x_{i,j}(t-1) - x_{i,j}(t)\Big)^+}{p_j},
\end{aligned}$$

where the first equality holds because of the Law of Total Expectation (Theorem 2.1) and the second equality follows from Property (ii) and the observation discussed above. □

Proof of Theorem 4.3. We first show that the above described algorithm incurs a total cost of at most $3\sum_{j=1}^{n_S} c_j$ while maintaining a solution that has a small load on each machine in expectation. To this end, let $L_i(t) := \sum_{j:X_j(t)=i} p_j$ denote the random load on machine i at time t. We start with showing that $\mathbb{E}[L_i(t)] \leq 6C^*_{\max}$ for all $1 \leq i \leq m$. Since a bound on the expected load per machine is not sufficient to bound the expectation of the maximum load, i.e., $\mathbb{E}\big[\max_i L_i(t)\big]$, afterwards, we show how to guarantee a makespan less than $18C^*_{\max}$ with probability one at the loss of another constant factor in the reassignment cost.

With Lemma 4.4, it follows

$$\mathbb{E}\big[L_i(t)\big] = \sum_{j=1}^{t} \mathbb{P}[X_j(t) = i]p_j = \sum_{j=1}^{t} \frac{x_{i,j}(t)}{p_j} p_j = l_i(t),$$

where $l_i(t)$ is the fractional load on machine i after having assigned job t. By Corollary 4.2, we know that $\max_{1\leq i\leq m} l_i(t) \leq 6C^*_{\max}$ if there exists a feasible solution with makespan $C^*_{\max}$. Now consider the reassignment cost $\tilde{c}(t)$ our algorithm incurs over the course of the arrival of t small jobs. For $1 \leq j \leq t$, the algorithm pays c_j whenever $X_j(t-1) \neq X_j(t)$. Thus, with Property (iii) of Lemma 4.4, we have

$$\mathbb{E}\big[\tilde{c}(t)\big] = \sum_{j=1}^{t} \mathbb{P}[X_j(t-1) \neq X_j(t)]c_j$$

$$
\begin{aligned}
&= \sum_{j=1}^{t} \sum_{\substack{i\in[m]:\\ \mathbb{P}[X_j(t-1)=i]>0}} \frac{c_j}{p_j}(x_{i,j}(t) - x_{i,j}(t-1))^+ \\
&\leq \sum_{j=1}^{t}\sum_{i=1}^{m} \frac{c_j}{p_j}|x_{i,j}(t) - x_{i,j}(t-1)| \\
&= c(t).
\end{aligned}
$$

Again, with Corollary 4.2, the expected total cost of the randomized algorithm is bounded by $\sum_{t=1}^{n} c(t) \leq 3\sum_{j=1}^{n} c_j$.

Unfortunately, bounding $\mathbb{E}\big[\, L_i(t)\,\big]$ does not imply a bound on $\mathbb{E}\big[\max_{1\leq i\leq m} L_i(t)\,\big]$ as noted by [GKS14]. Indeed, a simple balls into bins argument shows that even though the expected load of each machine is at most a constant, the expected maximum of the loads is $\Omega\Big(\frac{\log m}{\log\log m}\Big)$.

We use the fact that we are only considering small jobs in order to get a better bound. Consider a time t as well as a machine i. Define the random variable $Y_{i,j}(t)$ to indicate whether or not j is assigned to i at time t. So, $Y_{i,j} = \mathbb{1}_{\{X_j(t)=i\}}$ and $L_i(t) = \sum_{j\in\mathcal{J}_S} p_j Y_{i,j}$. We have $\mathbb{E}\Big[\sum_{j\in\mathcal{J}_S} p_j Y_{i,j}(t)\Big] = l_i(t) \leq 6C^*_{\max}(t)$ as discussed above. Here, $C^*_{\max}(t)$ denotes the optimal makespan in round t. Now, we bound the probability that the makespan of our schedule exceeds $18C^*_{\max}(t)$ in round t. We start by giving a union bound on this probability

$$
\begin{aligned}
\mathbb{P}\Big[\max_i l_i(t) \geq 18C^*_{\max}(t)\Big] &= \mathbb{P}\left[\exists i \,:\, \sum_{j\in\mathcal{J}_S: X_j(t)=i} p_j \geq 18C^*_{\max}(t)\right] \\
&\leq \sum_{i=1}^{m} \mathbb{P}\left[\sum_{j\in\mathcal{J}_S: X_j(t)=i} p_j \geq 18C^*_{\max}(t)\right] \\
&= \sum_{i=1}^{m} \mathbb{P}\left[\sum_{j\in\mathcal{J}_S:} \gamma\frac{Y_{i,j}(t)p_j}{C^*_{\max}(t)} \geq 18\gamma\right].
\end{aligned}
$$

Fix a machine i and a round t and observe that the random variables $\gamma\frac{Y_{i,j}(t)p_j}{C^*_{\max}(t)}$ are independently distributed in $[0,1]$ with $\mathbb{E}\Big[\sum_{j\in\mathcal{J}_S}\gamma\frac{Y_{i,j}(t)p_j}{C^*_{\max}(t)}\Big] = \gamma\frac{l_i(t)}{C^*_{\max}(t)} \leq 6\gamma$. Applying the Chernoff-Hoeffding bound (Theorem 2.3) with $\varepsilon = 18\frac{C^*_{\max}(t)}{l_i(t)} - 1$ and thus $\varepsilon^2 \geq 18\frac{C^*_{\max}(t)}{l_i(t)}$ yields

$$
\mathbb{P}\left[\sum_{j\in\mathcal{J}_S:} \gamma\frac{Y_{i,j}(t)p_j}{C^*_{\max}(t)} \geq 18\gamma\right] \leq \exp\left(-\varepsilon^2\frac{l_i(t)\gamma}{3C^*_{\max}(t)}\right) \leq \exp(-6\gamma) \leq \frac{1}{(mt)^6}.
$$

Inserting this in the bound calculated above gives

$$
\mathbb{P}\Big[\max_i l_i(t) \geq 18C^*_{\max}(t)\Big] \leq \frac{1}{m^5t^6}.
$$

Hence, for one instance with n jobs, the probability that the makespan of our algorithm

exceeds $18C^*_{\max}(t)$ in some round t is bounded by

$$\mathbb{P}\Big[\exists t : \max_i l_i(t) \geq 18C^*_{\max}(t)\Big] \leq \frac{1}{m^5}\sum_{t=1}^{n_S}\frac{1}{t^6} \leq \frac{1}{m^5}\frac{\pi^6}{945} \leq \frac{1.02}{m^5}.$$

Hence, whenever the randomized rounding algorithm incurs a makespan more than $18C^*_{\max}$, we just restart the algorithm from scratch and fast-forward to time t. Then, we reassign all small jobs accordingly incurring a reassignment cost of at most $C := \sum_{j\in\mathcal{J}_S} c_j$. If we observe such a failure mode, we run the algorithm independently of all previous runs. Hence, the probability that we observe such a failure mode k times for one instance is bounded by $\left(\frac{1.02}{m^5}\right)^k \leq \frac{1}{2^k}$ for $m \geq 2$. Thus, the expected cost of possible failure modes is bounded by

$$\sum_{k=1}^{\infty} C\,k\left(\frac{1}{2}\right)^k = 2C$$

if $m \geq 2$. We conclude that the algorithm is $\mathcal{O}(1)$-competitive in expectation with expected reassignment cost at most $\mathcal{O}(1)\sum_{j\in\mathcal{J}_S} c_j$ when combined with the failure mode. □

4.3.3 Arbitrary Jobs

The main result of this section is the following theorem.

Theorem 4.5. *There is a randomized online algorithm maintaining an assignment with expected makespan at most $\mathcal{O}(\log\log mn)C^*_{\max}$ while incurring an expected reassignment cost of at most $\mathcal{O}(1)\sum_{j=1}^{n} c_j$, where $C^*_{\max}$ is the optimal makespan.*

The proof of the theorem follows the idea in [GKS14] for the proof of their Theorem 8.1. Consider an arbitrary job set $\mathcal{J}$ and let $p'_j := 2^{\lfloor\log p_j\rfloor}$. Define $p'_{i,j} = p'_j$ if $p_{i,j} = p_j$ and $p'_{i,j} = \infty$ otherwise. By a fairly standard argument, this implies that the optimal makespan of the original instance is at most twice the optimal makespan of the modified instance. Hence, at the loss of an additional factor 2 in the competitive ratio, we assume from now on that the processing times are powers of two and say job j belongs to *class* $\mathcal{C}_k$ if $p_j \in [2^{k-1}, 2^k)$ for $k \in \mathbb{N}$.

For simplicity, let us start with supposing that we know n, the number of jobs we will encounter, and $C^*_{\max}$, the optimal makespan. Based on these two values, we classify each arriving job as big or as small. We use $\mathcal{J}_B$ and $\mathcal{J}_S$ to refer to these two types of jobs. Let $\gamma = \log mn$. We say a job j is *big* if $p_j \geq \frac{C^*_{\max}}{\gamma}$, and otherwise, the job is *small*. As jobs may only arrive, each job makes the transition from big to small at most once. Hence, using again a guess-and-double framework for $C^*_{\max}$ and the current value of n enables us to justify this assumption.

Our algorithm treats these jobs differently: A small job j is assigned by the randomized algorithm described in Section 4.3.2

For big jobs, we use the partitioning into classes $\mathcal{C}_k$ and consider each class separately. The rounding of the processing times upon arrival implies that jobs in the same class have the same processing time, and, thus, we obtain an instance of online load balancing with unit-size jobs by scaling the instance by 2^{k-1}.

Formally, after the arrival of the first job 1, we round down p_1 to the next power of 2 before setting $C^*_{\max} = 2p_1$. Given m, we additionally set $\gamma = \log m$ as $n = 1$ currently holds. With each new job, we update $\gamma = \log mn$.

Then, we classify each job as big job if its processing time is at least $\frac{C^*_{\max}}{\gamma}$, otherwise the job is small. Based on the type of job j, we run the algorithm for small jobs (Section 4.3.2), or we invoke the algorithm for unit-size jobs (Section 4.3.1) for class $\mathcal{C}_k$, where $k = \lfloor \log p_j \rfloor$. Whenever the shortest-augmenting-path algorithm used a class of big jobs reports that it cannot find a solution with makespan at most $3C^*_{\max}$ or the randomized algorithm for small jobs cannot find a solution with makespan at most $18C^*_{\max}$, we double $C^*_{\max}$. If a previously big job becomes small due to such an update of $\frac{C^*_{\max}}{\gamma}$, we treat this transition as a new arrival of a small job and invoke the algorithm for small jobs *before* assigning the new job j.

Before proving the main result of this chapter, we need to analyze the algorithm's performance when assigning big jobs. The rounding of the processing times upon arrival implies that jobs in the same class have the same processing time, and, thus, we obtain an instance of online load balancing with unit-size jobs by scaling the instance by 2^{k-1}. Since there are $\mathcal{O}(\log \log mn)$ classes of big jobs, Corollary 4.2 immediately yields the following result on big jobs.

Corollary 4.6. *There is an online algorithm maintaining an assignment of the big jobs $\mathcal{J}_B$ with expected makespan at most $\mathcal{O}(\log \log mn)C^*_{\max}$ and reassignment cost at most $\mathcal{O}(1)\sum_{j\in\mathcal{J}_B} c_j$.*

Proof of Theorem 4.5. Theorem 4.3 guarantees that the algorithm maintains a schedule for the small jobs of makespan at most $\mathcal{O}(1)C^*_{\max}$ while incurring a reassignment cost of at most $\mathcal{O}(1)\sum_{j\in\mathcal{J}_S} c_j$. Corollary 4.6 implies that the schedule for the big jobs has makespan at most $\mathcal{O}(\log \log mn)C^*_{\max}$ with total cost bounded by $\mathcal{O}(1)\sum_{j\in\mathcal{J}_B} c_j$. Hence, the algorithm achieves a makespan of at most $\mathcal{O}(\log \log mn)C^*_{\max}$ with total reassignment cost at most $\mathcal{O}(1)\sum_{j\in\mathcal{J}} c_j$. □

4.4 Concluding Remarks

In this chapter, we design an online algorithm for load balancing with reassignment cost. Somewhat surprisingly, the competitive ratios achieved in all three reassignment models is equal (up to constants). It remains an interesting open question whether the problem admits a constant competitive algorithm in any reassignment model with constant reassignment factor or if there even exists an online algorithm allowing for tradeoff between the competitive ratio and reassignment factor. We would like to point out that the analysis of the algorithm is tight, and thus for affirmatively answering these questions one needs to design a new algorithm.

Further interesting research directions include maximizing the minimal load and considering the fully dynamic setting where items might leave as well. The difficulty in both settings is that there might be time points where the optimum is equal to 0 which makes these types of problems notoriously difficult for approximation. One way to overcome these difficulties would be to aim for competitive ratios with an additive constant; such an approach is developed, e.g., in [BRVW20], for online load balancing on identical machines.

5

Online Throughput Maximization

We study an online scheduling problem where jobs with deadlines arrive online over time at their release dates, and the task is to determine a preemptive schedule on m machines which maximizes the number of jobs that complete before their deadline. To circumvent known impossibility results, we make a standard slackness assumption by which the feasible time window for scheduling a job is at least $1+\varepsilon$ times its processing time, for some $\varepsilon > 0$. We design a simple admission scheme that achieves a competitive ratio of $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$. This is best possible as our matching lower bound shows.
On the technical side, we develop a combinatorial tool for analyzing the competitive ratio of a certain class of non-migratory online algorithms. As the next chapter shows, this tool is also of interest in a closely related problem.
Bibliographic Remark: The lower bound and an early version of the algorithm for one machine as well as parts of its analysis are based on joint work with L. Chen, N. Megow, K. Schewior, and C. Stein [CEM+20]. The generalization to multiple machines and the remaining parts of the analysis are based on joint work with N. Megow and K. Schewior [EMS20]. Therefore, some parts correspond to or are identical with [CEM+20] and [EMS20].

Table of Contents

5.1 Introduction

We consider a model in which jobs arrive online over time at their *release date* $r_j \geq 0$. Each job has a *processing time* $p_j \geq 0$, and a *deadline* $d_j \geq 0$. In order to complete, a job must receive a total of p_j units of processing time in the interval $[r_j, d_j)$. There are m identical machines to schedule jobs. We allow *preemption*, that is, the processing of a job can be interrupted and resumed at some later point in time. Further, we distinguish *migratory* and *non-migratory* algorithms. If an algorithm is migratory or it is allowed to use *migration*, then any preempted job may resume processing on any machine while it may only be completed by the machine it first started on otherwise. In a feasible schedule, no job is run in parallel with itself and no machine processes more than one job at any time. If a schedule completes a set S of jobs, then the cardinality $|S|$ of the set S is its *throughput*, which has to be maximized.

We analyze the performance of algorithms using standard *competitive analysis* [ST85] in which the performance of an algorithm is compared to that of an optimal offline algorithm with full knowledge of the future. More precisely, an online algorithm $\mathcal{A}$ is called c-competitive if it achieves for any input instance $\mathcal{I}$ a total value of $|\mathcal{A}(\mathcal{I})| \geq \frac{1}{c}|\textsc{Opt}(\mathcal{I})|$, where $\textsc{Opt}(\mathcal{I})$ is the set of jobs scheduled by an optimal (offline) algorithm and $\mathcal{A}(\mathcal{I})$ the one scheduled by $\mathcal{A}$.

The problem becomes hopeless when preemption is not allowed: whenever an algorithm starts a job j without being able to preempt it, it may miss the deadlines of an arbitrary number of jobs that would have been schedulable if j had not been started. Therefore, we focus on *preemptive online* throughput maximization.

Hard examples for online algorithms tend to involve jobs that arrive and then *must* immediately be processed since $d_j - r_j \approx p_j$. It is entirely reasonable to bar such jobs from a system, requiring that any submitted job contains some *slack*. That is, we must have some separation between p_j and $d_j - r_j$. To this end, we say that an instance has *ε-slack* if every job satisfies $d_j - r_j \geq (1+\varepsilon)p_j$. We develop an algorithm whose competitive ratio depends on ε; the greater the slack, the better we expect the performance of our algorithm to be. This slackness parameter captures certain aspects of Quality-of-Service provisioning and admission control, see, e.g., [GGP97, LWF96], and it has been considered in previous work, e.g., in [AKL+15, BH97, GNYZ02, Gol03, LMNY13, SS16]. Other results for scheduling with deadlines use speed scaling, which can be viewed as another way to add slack to the schedule; see, e.g., [ALLM18, BCP11, IM18, PS10].

Related work Preemptive online scheduling and admission control have been studied rigorously. There are several results regarding the impact of deadlines on online scheduling; see, e.g., [BHS94, GNYZ02, Gol03] and references therein.

For maximizing the throughput on a single machine, Baruah, Haritsa, and Sharma [BHS94] show that, in general, no online algorithm achieves a bounded competitive ratio. Thus, their result justifies our assumption on ε-slackness of each job. Moreover, they consider special

cases such as unit-size jobs or agreeable deadlines where they provide constant-competitive algorithms even without further assumptions on the slack of the jobs. Here, deadlines are agreeable if $r_j \leq r_{j'}$ for two jobs j and j' implies $d_j \leq d_{j'}$. Despite the strong impossibility results for general instances, Kalyanasundaram and Pruhs [KP03] give a *randomized* $\mathcal{O}(1)$-competitive algorithm. No deterministic algorithm has been known prior to our $\Theta\left(\frac{1}{\varepsilon}\right)$-competitive algorithm [CEM+20].

When the scheduler is concerned with machine utilization, i.e., she wants to maximize the total processing time of completed jobs, the problem becomes more tractable. On a single machine, Baruah et al. [BKM+91, BKM+92] provide a best-possible online algorithm achieving a competitive ratio of 4, even without any slackness assumptions. Baruah and Haritsa [BH97] are the first to investigate the problem under the assumption of ε-slack and give a $\frac{1+\varepsilon}{\varepsilon}$-competitive algorithm which is asymptotically best possible. For parallel machines (though without migration), DasGupta and Palis [DP00] show that a simple greedy algorithm achieves the same performance guarantee of $\frac{1+\varepsilon}{\varepsilon}$ and give an asymptotic matching lower bound. Schwiegelshohn and Schwiegelshohn [SS16] show that migration helps the online algorithm and improve the competitive ratio to $\sqrt[m]{\frac{1}{\varepsilon}}$ for m machines. We emphasize that this result is in contrast to our results as our non-migratory algorithm is also $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$-competitive in the migratory setting.

For maximizing the *weight* of the completed jobs, Lucier et al. [LMNY13] give an $\mathcal{O}\left(\frac{1}{\varepsilon^2}\right)$-competitive algorithm in the most general weighted setting. Prior to considering slackness, Baruah et al. [BKM+91] showed a lower bound of $\frac{1}{(1+\sqrt{k})^2}$ for any deterministic single-machine online algorithm, where $k = \frac{\max_j w_j/p_j}{\min_j w_j/p_j}$ is the *importance ratio* of a given instance. Koren and Shasha give a matching upper bound [KS95] and generalize it to $\Theta(\ln k)$ for parallel machines if $k > 1$ [KS94].

Our contribution We give an $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$-competitive online algorithm, the *threshold algorithm*, for maximizing throughput on parallel identical machines. As we originally developed this algorithm for a more general setting, we considerably simplify the exposition when compared to the algorithm in [CEM+20]. We observe that, due to this simplification, the single-machine variant of our algorithm is now identical (up to constants) to the algorithm developed by Lucier et al. [LMNY13] for maximizing the weighted throughput. On parallel machines, the algorithms are closely related although we do not need to select the machine a job is assigned to as carefully as they do. Our tight analysis shows that this algorithm is $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$-competitive for maximizing the throughput. In contrast to the analysis in [LMNY13] based on dual fitting, we give a purely combinatorial analysis. We also prove that our algorithm is *optimal* by giving a matching lower bound (ignoring constants) for any deterministic online algorithm.

As a key contribution on the technical side, we give a strong bound on the processing volume of any feasible non-migratory schedule in terms of the accepted volume of a certain class of online algorithms. It is crucial for our analysis and might be of independent interest.

5.2 The Threshold Algorithm

In this section, we present our algorithm for online throughput maximization. Further, we state the main result and provide a road map for its proof. We assume that an online algorithm is given the slackness constant $\varepsilon > 0$.

5.2.1 The Threshold Algorithm

To gain some intuition for our algorithm, we first describe informally the underlying design principles. The threshold algorithm never migrates jobs between machines. In other words, a job is only processed by the machine it initially was started on. We say the job has been *admitted* to this machine. Moreover, a running job can only be preempted by significantly smaller-size jobs, i.e., smaller by a factor of at least $\frac{\varepsilon}{4}$ with respect to the processing time, and a job j cannot start for the first time when its remaining slack is too small, i.e., less than $\frac{\varepsilon}{2}p_j$.

We note that the algorithm developed in [LMNY13] also follows these design principles: It only admits jobs that are smaller by a factor of γ, the *threshold parameter*, with respect to the processing time of the currently running job. Second, it only starts jobs for the first time if the remaining slack is at least $\mu - 1$, where μ is the *gap parameter*. By setting $\gamma = \frac{\varepsilon}{4}$ and $\mu = 1 + \frac{\varepsilon}{2}$, we essentially recover the algorithm developed in [LMNY13]. For the sake of self-containment, we give a formal description of the threshold algorithm adapted to our setting.

At any time τ, the threshold algorithm maintains two sets of jobs: *admitted jobs*, which have been started before or at time τ, and *available jobs*. A job j is available if it is released before or at time τ, is not yet admitted, and τ is not too close to its deadline, i.e., $r_j \leq \tau$ and $d_j - \tau \geq \left(1 + \frac{\varepsilon}{2}\right)p_j$. The intelligence of the threshold algorithm lies in how it admits jobs. The actual scheduling decision then is simple and independent of the admission of jobs: at any point in time and on each machine, schedule the shortest job that has been admitted to this machine and has not completed its processing time. In other words, we schedule admitted jobs on each machine in Shortest Processing Time (SPT) order. The threshold algorithm never explicitly considers deadlines except when deciding whether to admit jobs. In particular, jobs can even be processed after their deadline.

At any time τ, when there is a job j available and a machine i *idle*, i.e., i is not processing any previously admitted job j', the shortest available job $j^\star$ is immediately admitted to machine i at time $a_{j^\star} := \tau$. There are two events that trigger a decision of the threshold algorithm: the release of a job and the completion of a job. If one of these events occurs at time τ, the threshold algorithm invokes the preemption subroutine. This routine iterates over all machines and compares the processing time of the smallest *available* job $j^\star$ with the processing time of job j that is currently scheduled on machine i. If $p_{j^\star} < \frac{\varepsilon}{4}p_j$, job $j^\star$ is admitted to machine i at time $a_{j^\star} := \tau$ and, by the above scheduling routine, immediately starts processing. We summarize the threshold algorithm in Algorithm 5.1.

Algorithm 5.1: Threshold algorithm

Scheduling routine: At any time τ and on any machine i, run the job with shortest processing time that has been admitted to i and has not yet completed.

Event: Upon release of a new job at time τ:
Call threshold preemption routine.

Event: Upon completion of a job j at time τ:
Call threshold preemption routine.

Threshold preemption routine:
$j^\star \leftarrow$ a shortest available job at τ, i.e., $j^\star \in \arg\min\{p_j \mid j \in \mathcal{J}, r_j \leq \tau \text{ and } d_j - \tau \geq (1+\frac{\varepsilon}{2})p_j\}$
$i \leftarrow 1$
while $j^\star$ is not admitted and $i \leq m$ do
  $j \leftarrow$ job processed on machine i at time τ
  if $j = \emptyset$ do
    admit job $j^\star$ to machine i
    call threshold preemption routine
  else-if $p_{j^\star} < \frac{\varepsilon}{4}p_j$ do
    admit job $j^\star$ to machine i
    call threshold preemption routine
  else
    1. $i \leftarrow i+1$

5.2.2 Main Result and Road Map of the Analysis

In the analysis we focus on instances with small slack as they constitute the hard case. Note that instances with large slack clearly satisfy a small-slack assumption. In such a case, we simply run our algorithm by setting $\varepsilon = 1$ and obtain constant-competitive ratios. Therefore, we assume for the remainder that $0 < \varepsilon \leq 1$.

Theorem 5.1. *Let $0 < \varepsilon \leq 1$. The threshold algorithm is $\Theta\left(\frac{1}{\varepsilon}\right)$-competitive for online throughput maximization.*

This is an improvement by a factor $\frac{1}{\varepsilon}$ upon the best previously known upper bound [LMNY13] (given for weighted throughput).

Road map During the analysis, we use the fact that our algorithm never migrates jobs. In the analysis, we first compare the throughput of our algorithm to the solution of an optimal non-migratory schedule. We then use a well-known result by Kalyanasundaram and Pruhs [KP01] to compare this to an optimal solution that may exploit migration. Here, ω_m is the maximal ratio of the throughput of an optimal migratory schedule to the throughput of an optimal non-migratory schedule.

Theorem 5.2 (Theorem 1.1 in [KP01]). $\omega_m \leq \frac{6m-5}{m}$.

For relating the throughput of the threshold algorithm to the throughput of an optimal (non-migratory) schedule, we rely on a key design principle of the threshold algorithm, which

is that, whenever the job set admitted to a machine is fixed, the scheduling of the jobs follows the simple SPT order. This enables us to split the analysis into two parts.

In the first part, we argue that the scheduling routine can handle the admitted jobs sufficiently well. That is, an adequate number of the admitted jobs is completed on time; see Section 5.3. Here, we use again that the threshold algorithm is non-migratory and consider each machine individually.

For the second part, we observe that the potential admission of a new job $j^\star$ to machine i is solely based on its availability and on its size relative to j_i, the job currently processed by machine i. More precisely, given the availability of $j^\star$, if $p_{j^\star} < \frac{\varepsilon}{4} p_{j_i}$ and i is the first machine with this property, then $j^\star$ is admitted to machine i. This implies that $\frac{\varepsilon}{4}$ times the maximum of the processing times of the jobs j_i acts as a *threshold*, and only available jobs with processing time less than this threshold qualify for admission by the threshold algorithm. Hence, any available job that the threshold algorithm does not admit has to violate the threshold.

Based on this observation, we develop a general charging scheme for *any* non-migratory online algorithm satisfying the property that, at any time τ, the algorithm maintains a time-dependent threshold and the shortest available job smaller than this threshold is admitted by the algorithm. We formalize this description and analyze the competitive ratio of such algorithms in Section 5.4 before applying this general result to our particular algorithm.

5.3 Successfully Completing Sufficiently Many Admitted Jobs

In this section, we show that the threshold algorithm completes half of all admitted jobs on time. Since the threshold algorithm is non-migratory, it suffices to consider each machine separately. We start by defining *interruption trees* to capture the intricate structure of the processing intervals. This enables us to construct a worst-case instance for the threshold algorithm where "worst" is with respect to the ratio between admitted and successfully completed jobs. The main result of this section is the following theorem.

Theorem 5.3. *Let $0 < \varepsilon \leq 1$. Then the threshold algorithm completes at least half of all admitted jobs before their deadline.*

Interruption trees To analyze the performance of the threshold algorithm on a given instance, we consider the final schedule per machine and investigate it retrospectively. Our analysis crucially relies on understanding the interleaving structure of the processing intervals that the algorithm constructs. This structure is due to the interruption by smaller jobs and can be captured well by a tree or forest in which each job is represented by one vertex. A job vertex is the child of another vertex if and only if the processing of the latter is interrupted by the first one. The leaves correspond to jobs with contiguous processing. We also add a machine job M_i for $i \in [m]$ with processing time ∞ and admission date $-\infty$. The children of machine

job M_i are all jobs admitted to machine i that did not interrupt the processing of another job. Thus, we can assume that the instance is represented by m trees which we call *interruption trees*. An example of an interruption tree is given in Figure 5.1.

Let $\pi(j)$ denote the *parent* of j. Further, let T_j be the subtree of the interruption tree rooted in job j and let the forest T_{-j} be T_j without its root j. By slightly abusing notation, we denote the tree/forest as well as the set of its job vertices by T_*.

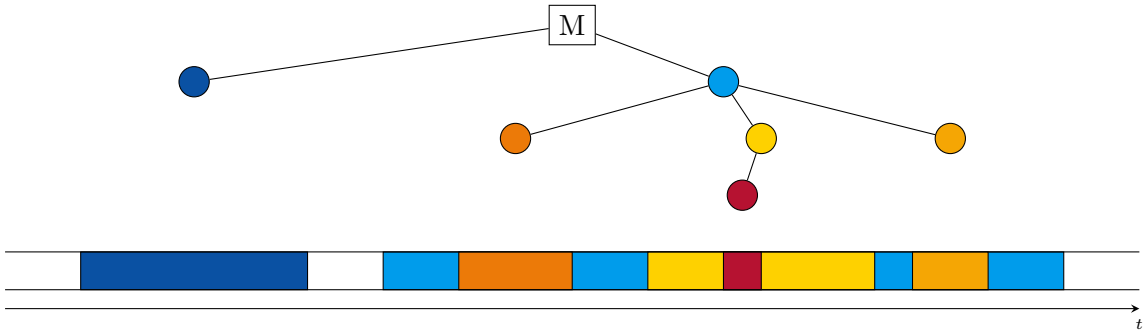

Figure 5.1: Gantt chart of a single-machine schedule generated by the threshold algorithm and the resulting interruption tree

Instance modifications The proof of Theorem 5.3 relies on two technical results that enable us to restrict to instances with one machine and further only consider jobs that are contained in the interruption tree created on this instance. We start with the following observation. Let $\mathcal{I}$ be an instance of online throughput maximization with the job set $\mathcal{J}$ and let $J \subseteq \mathcal{J}$ be the set of jobs admitted by the threshold algorithm at some point. It is easy to see that a job $j \notin J$ does not influence the scheduling or admission decisions of the threshold algorithm. The next lemma formalizes this statement and follows immediately from the just made observations.

Lemma 5.4. *For any instance $\mathcal{I}$ for which the threshold algorithm admits the job set $J \subseteq \mathcal{J}$, the reduced instance $\mathcal{I}'$ containing only the jobs J forces the threshold algorithm with consistent tie breaking to admit all jobs in J and to create the same schedule as produced for the instance $\mathcal{I}$.*

The proof of the main result compares the number of jobs finished on time, $F \subseteq J$, to the number of jobs unfinished by their respective deadlines, $U = J \setminus F$. To further simplify the instance, we use that the threshold algorithm is non-migratory and restrict to single-machine instances. To this end, let $F^{(i)}$ and $U^{(i)}$ denote the finished and unfinished, respectively, jobs on machine i.

Lemma 5.5. *Let $i \in [m]$. There is an instance $\mathcal{I}'$ on one machine with job set $\mathcal{J}' = F^{(i)} \cup U^{(i)}$. Moreover, the schedule of the threshold algorithm for instance $\mathcal{I}'$ with consistent tie breaking is identical to the schedule of the jobs $\mathcal{J}'$ on machine i. In particular, $F' = F^{(i)}$ and $U' = U^{(i)}$.*

Proof. By Lemma 5.4, we can restrict to the jobs admitted by the threshold algorithm. Hence, let $\mathcal{I}$ be such an instance with $F^{(i)} \cup U^{(i)}$ being admitted to machine i. As the threshold algorithm is non-migratory, the sets of jobs scheduled on two different machines are disjoint. Let $\mathcal{I}'$ consist of the jobs in $\mathcal{J}' := F^{(i)} \cup U^{(i)}$ and one machine. The threshold algorithm on instance $\mathcal{I}$ admits all jobs in $\mathcal{J}$. In particular, it admits all jobs in $\mathcal{J}'$ to machine i.

We inductively show that the schedule for the instance $\mathcal{I}'$ is identical to the schedule on machine i in instance $\mathcal{I}$. To this end, we index the jobs in $\mathcal{J}'$ in increasing admission time points in instance $\mathcal{I}$.

It is obvious that job $1 \in \mathcal{J}'$ is admitted to the single machine at its release date r_1 as happens in instance $\mathcal{I}$ since the threshold algorithm uses consistent tie breaking. Suppose that the schedule is identical until the admission of job $j^\star$ at time $a_{j^\star} = \tau$. If $j^\star$ does not interrupt the processing of another job, then $j^\star$ will be admitted at time τ in $\mathcal{I}'$ as well. Otherwise, let $j \in \mathcal{J}'$ be the job that the threshold algorithm planned to process at time τ *before* the admission of job $j^\star$. Since $j^\star$ is admitted at time τ in $\mathcal{I}$, $j^\star$ is available at time τ, satisfies $p_{j^\star} < \frac{\varepsilon}{4} p_j$, and did not satisfy both conditions at some earlier time τ' with some earlier admitted job j'. Since the job set in $\mathcal{I}'$ is a subset of the jobs in $\mathcal{I}$ and we use consistent tie breaking, no other job $j^* \in \mathcal{J}'$ that satisfies both conditions is favored by the threshold algorithm over $j^\star$. Therefore, job $j^\star$ is also admitted at time τ by the threshold algorithm in instance $\mathcal{I}'$. Thus, the schedule created by the threshold algorithm for $\mathcal{J}'$ is identical to the schedule of $\mathcal{J}$ on machine i in the original instance. □

We want to show that the existence of a job j that finishes after its deadline implies that the subtree T_j rooted in j contains more finished than unfinished jobs. To this end, we prove a stronger statement about the number of finished and unfinished jobs in any subtree T_j based on the length of the interval $[a_j, C_j)$ where a_j is again the admission date of job j and C_j is its completion time. We want to analyze the schedule generated by the threshold algorithm in the interval $[a_j, C_j)$, i.e., the schedule of the jobs in T_j. Let F_j denote the set of jobs in T_j that *finish on time*. Similarly, we denote the set of jobs in T_j that complete after their deadlines, i.e., that are *unfinished at their deadline*, by U_j; we call these jobs *unfinished* for simplification throughout the proof.

Lemma 5.6. *If $C_j - a_j \geq (\beta + 1)p_j$ for $\beta > 0$, then $|F_j| - |U_j| \geq \left\lfloor \frac{4\beta}{\varepsilon} \right\rfloor$.*

To prove this lemma, we further restrict the instances we need to consider. Lemma 5.4, Lemma 5.5, and the next lemma justify this restriction. After restricting to single-machine instances and excluding all jobs not contained in the interruption tree for this machine, we exploit the special structure of the schedule generated by the threshold algorithm when proving the next lemma. To this end, we introduce some notation to talk about the position of a particular job in the tree relative to the root of the tree. More precisely, we define the *height* of an interruption tree to be the edge-length of a longest path from root to leaf and the *height* of the node j in the tree to be the height of T_j.

Lemma 5.7. *Let j be a job in the interruption tree with $C_j - a_j \geq (\beta+1)p_j$ and $|F_j| - |U_j| < \left\lfloor \frac{4\beta}{\varepsilon} \right\rfloor$. There exists an instance $\mathcal{I}'$ with $|F'_j| - |U'_j| = |F_j| - |U_j|$ and an unfinished height-1 job $j^\star$ in $\mathcal{I}'$ satisfying the following properties.*

(P1) No job is admitted in $[d_{j^\star}, C_{j^\star})$.

(P2) The union of the processing intervals of the children of $j^\star$ is an interval.

Before proving this lemma, we use it to show Lemma 5.6.

Proof of Lemma 5.6. Toward a contradiction, suppose that there is an instance such that there is a job j admitted by the threshold algorithm with $C_j - a_j \geq (\beta+1)p_j$ and $|F_j| - |U_j| < \left\lfloor \frac{4\beta}{\varepsilon} \right\rfloor$. Among all such instances let $\mathcal{I}$ be an instance with the minimal number of jobs.

The goal is to construct an instance $\mathcal{I}'$ that satisfies $C_j - a_j \geq (\beta+1)p_j$ and $|F_j| - |U_j| < \left\lfloor \frac{4\beta}{\varepsilon} \right\rfloor$ although it uses fewer jobs than $\mathcal{I}$. By Lemma 5.7, we can assume without loss of generality that the instance $\mathcal{I}$ satisfies the Properties (P1) and (P2). These assumptions enable us to create a new instance $\mathcal{I}'$ that merges three jobs to one larger job without violating $C_j - a_j \geq (\beta+1)p_j$ or increasing $|F_j| - |U_j|$. The three jobs will be leaves with the same (unfinished) parent $j^\star$ in T_j. In fact, if $j^\star$ is an unfinished job, then $C_{j^\star} - a_{j^\star} \geq (1 + \frac{\varepsilon}{2})p_{j^\star}$. Any job k that may postpone $j^\star$ satisfies $p_k < \frac{\varepsilon}{4}p_{j^\star}$. Hence, if the children of $j^\star$ are all leaves, there exist at least three jobs that interrupt $j^\star$.

To this end, consider an unfinished job $j^\star$ as in Lemma 5.7. The modification has three steps. In the first step, we merge three jobs in $T_{-j^\star}$. In the second step, we replace $j^\star$ by a similar job j^* to ensure that the instance still satisfies the ε-slack assumption. In the third step, we adapt jobs $k \notin T_{-j^\star}$ to guarantee that j^* is admitted at the right point in time. Then, we show that the resulting instance still satisfies $|F_j| - |U_j| < \left\lfloor \frac{4\beta}{\varepsilon} \right\rfloor$ and $C_j - a_j \geq (\beta+1)p_j$. Parts of the instance $\mathcal{I}$ and $\mathcal{I}'$ are shown in Figure 5.2.

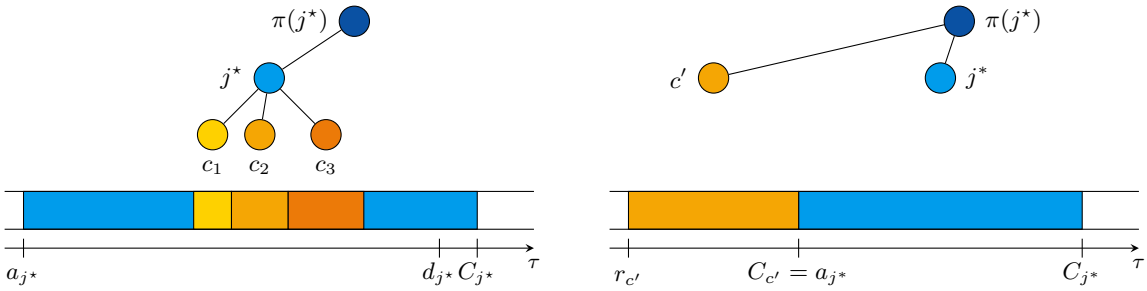

Figure 5.2: Modifications to obtain instance $\mathcal{I}'$ in the proof of Lemma 5.6. The deadline of job j^* satisfies $d_{j^*} > C_{j^8}$ and is not shown anymore.

Since $j^\star$ is admitted at $a_{j^\star} \leq d_{j^\star} + (1 + \frac{\varepsilon}{2})p_{j^\star}$ and not finished by the threshold algorithm on time, $C_{j^\star} - a_{j^\star} \geq (1 + \frac{\varepsilon}{2})p_{j^\star}$. Any job that may postpone $j^\star$ satisfies $p_k < \frac{\varepsilon}{4}p_{j^\star}$. Hence, there have to be at least three jobs that interrupt $j^\star$. Among these, consider the first three

jobs c_1, c_2, and c_3 (when indexed in increasing order of release dates). We create a new instance by deleting c_1, c_2, and c_3 and adding a new job c' such that c' is released at the admission date of $j^\star$ in $\mathcal{I}$ and it merges c_1, c_2, and c_3, i.e.,

$$r_{c'} := a_{j^\star}, \qquad p_{c'} := p_{c_1} + p_{c_2} + p_{c_3}, \qquad \text{and} \qquad d_{c'} := r_{c'} + (1+\varepsilon)p_{c'}.$$

Second, we replace $j^\star$ by a new job j^* that is released at $r_{j^*} := a_{j^\star} + p_{c'}$, has the same processing time, i.e., $p_{j^*} = p_{j^\star}$, and has a deadline $d_{j^*} := \max\{d_{j^\star}, r_{j^*} + (1+\varepsilon)p_{j^*}\}$.

In the third step of our modification, we replace every job k with $r_k \in [r_{j^\star}, a_{j^\star}]$ and $p_k \leq p_{j^\star}$ by a new job k' that is released slightly after j^*, i.e., $r_{k'} := r_{j^*} + \varrho$ for $\varrho > 0$. More precisely, we choose ϱ such that $\varrho < \left(1 - \frac{\varepsilon}{2}\right)p_k$ for each job k that is subjected to this modification. It is important to note that we do not change the processing time or the deadline of k', i.e., $p_{k'} = p_k$ and $d_{k'} = d_k$. This ensures that k' finishes on time if and only if k finishes on time. This modification is feasible, i.e., $d_{k'} - r_{k'} \geq (1+\varepsilon)p_{k'}$, because of two reasons. First,

$$C_{j^\star} - r_{j^*} = C_{j^\star} - (a_{j^\star} + p_{c'}) = C_{j^\star} - a_{j^\star} - (p_{c_1} + p_{c_2} + p_{c_3}) \geq p_{j^\star}$$

as c_1, c_2, and c_3 postponed $j^\star$ by their processing times in $\mathcal{I}$. Second, $d_k - C_{j^\star} \geq \left(1 + \frac{\varepsilon}{2}\right)p_k$ because we only consider jobs that are admitted at some point later than $C_{j^\star}$ by the threshold algorithm. Then,

$$d_{k'} - r_{k'} = d_k - C_{j^\star} + C_{j^\star} - r_{j^*} - \varrho \geq \left(1 + \frac{\varepsilon}{2}\right)p_k + p_{j^\star} - \varrho \geq \left(2 + \frac{\varepsilon}{2}\right)p_k - \varrho \geq (1+\varepsilon)p_{k'},$$

where the last but one inequality follows from the fact that only jobs with $p_k \leq p_{j^\star}$ are affected by the modification and the last inequality is due to the sufficiently small choice of ϱ.

So far, we have already seen that the resulting instance is still feasible. It is left to show that c' completes at $r_{c'} + p_{c'}$ as well as that j^* is admitted at r_{j^*} and it completes at $C_{j^\star}$.

Since it holds that $p_{c'} < \frac{3\varepsilon}{4}p_{j^\star} < p_{j^\star}$, the new job c' is the smallest available job at $a_{j^\star} = r_{c'}$ and any job that was interrupted by $j^\star$ is interrupted by c' as well. The jobs in $T_{-j^\star}$ are released one after the other by Property (P2) and $r_{c_1} > a_{j^\star}$. Thus, if j^* has at least one child c_4 left after the modification, it holds that $r_{c_4} = r_{c_1} + p_{c_1} + p_{c_2} + p_{c_3} = a_{j^\star} + p_{c'} + (r_{c_1} - a_{j^\star}) > r_{j^*}$. Hence, no remaining child is released in $[r_{c'}, r_{j^*}]$ in the modified instance. Any other job $k \in T_j$ released in $[r_{c'}, r_{j^*}]$ satisfies $p_k \geq \frac{\varepsilon}{4}p_{j^\star}$ as $k \notin T_{-j^\star}$. Because $p_{c'} < p_{j^\star}$, this implies that $p_k \geq \frac{\varepsilon}{4}p_{c'}$ holds as well, i.e., no such job k interrupts c'. Therefore, c' completes at r_{j^*}.

Job j^* is admitted at r_{j^*} if it is the smallest available job at that time. We have already seen that none of the remaining children of $j^\star$ is released in $[a_{j^\star}, r_{j^*}]$ that might prevent the threshold algorithm from admitting j^* at r_{j^*}. Furthermore, the third step of our modification guarantees that any job $k \in T_j \setminus T_{j^\star}$ that has processing time at most p_{j^*} is released after r_{j^*}. Therefore, j^* is the smallest available job at time r_{j^*} by construction, and it is admitted. As argued above, the modified instance is still feasible and the interval $[a_{\pi(j^\star)}, C_{\pi(j^\star)})$ is still the

interval of the schedule of jobs in $T_{\pi(j^\star)}$.

However, the second step of our modification might lead to $C_{j^*} \leq d_{j^*}$ which implies that j^* finishes on time while $j^\star$ does not finish on time. This changes the values of $|F_j|$ and $|U_j|$. Clearly, in the case that j^* completes before d_{j^*}, $|U'_j| = |U_j| - 1$. By a careful analysis, we see that in this case the number of finished jobs decreases by one as well because the three finished jobs c_1, c_2 and c_3 are replaced by only one job that finishes before its deadline. Formally, we charge the completion of c' to c_1, and the completion of j^* to c_2 which leaves c_3 to account for the decreasing number of finished jobs. Hence, $|F'_j| - |U'_j| = |F_j| - |U_j|$. If j^* does not finish by d_{j^*}, then $|F'_j| - |U'_j| = (|F_j| - 2) - |U_j|$. Therefore, the modified instance $\mathcal{I}'$ also satisfies $|F'_j| - |U'_j| < \left\lfloor \frac{4\beta}{\varepsilon} \right\rfloor$ but contains fewer jobs than $\mathcal{I}$ does. This is a contradiction. □

Now we proceed with proving Lemma 5.7.

Proof of Lemma 5.7. Let $\mathcal{I}$ be an instance and let job j be a job in its interruption tree satisfying $|F_j| - |U_j| < \left\lfloor \frac{4\beta}{\varepsilon} \right\rfloor$ and $C_j - a_j > (\beta + 1)p_j$. The construction of the instance $\mathcal{I}'$ consists of several modifications that maintain $|F'_j| - |U'_j| < \left\lfloor \frac{4\beta}{\varepsilon} \right\rfloor$ and $C_j - a_j > (\beta + 1)p_j$ without increasing the number of jobs compared to $\mathcal{I}$. We use $*'$ to refer to the object in the modified instance $\mathcal{I}'$ corresponding to $*$ in $\mathcal{I}$. The modifications are shown in Fig. 5.3. By Lemmas 5.4 and 5.5, we can assume that there is only one machine in this instance and that the instance $\mathcal{I}$ contains only the jobs admitted by the threshold algorithm. We start by showing the following three claims.

(C1) The instance contains at most $|T_j| + 1$ jobs.

(C2) The height of the interruption tree T'_j is at least two.

(C3) Each job of height one is unfinished.

Ad (C1) We observe that $p_k < p_j$ for any job $k \in T_{-j}$. Hence, $a_j < r_k$ for any such job by definition of the threshold algorithm. We distinguish two cases to prove the claim.

If $d_j - a_j \geq (1+\varepsilon)p_j$, we set $r'_j = a_j$, $p'_j = p_j$, and $d'_j = d_j$ and do not modify the remaining jobs in T_{-j} to define the set T'_j. Then, we set $\mathcal{J}' = T'_j$ to create a new feasible instance $\mathcal{I}'$ with slack ε. Clearly, the schedule produced by the threshold algorithm on instance $\mathcal{I}'$ is identical to the schedule of the threshold algorithm for instance $\mathcal{I}$ in the interval $[a_j, C_j)$ when using consistent tie breaking.

If $d_j - a_j < (1 + \varepsilon)p_j$, then $r_j < a_j$. Therefore, we set $r'_j = d_j - (1 + \varepsilon)p_j$ and do not change any other parameter of the jobs in T_j to obtain the set T'_j. The modified instance $\mathcal{I}'$ consists of the jobs in T'_j plus one additional job 0 to ensure that the threshold algorithm indeed produces the same schedule in the interval $[a_j, C_j)$ in both instances. More precisely, let $r'_0 = d_j - (1 + \varepsilon)p_j$, $p'_0 = (1 + \varepsilon)p_j - (d_j - a_j)$, and $d'_0 = r'_0 + (1 + \varepsilon)p'_0$. As $p'_0 < p_j = p'_j$, the threshold algorithm admits job 0 at r'_j and finishes this job at time $r'_j + p'_0 = a_j$. Thus, the

threshold algorithm admits job j also in instance $\mathcal{I}'$ at time a_j as it is the smallest available job. Since the remaining jobs have the same parameters in both instances, the schedules produced by the threshold algorithm for the interval $[a_j, C_j)$ are identical.

Ad (C2) We show that the height of T_j is at least two. Toward a contradiction, suppose that T_j is a star centered at j. Since any leaf finishes by definition of the threshold algorithm, the root j is the only job that could possibly be unfinished. As $|F_j| - |U_j| \leq \left\lfloor \frac{4\beta}{\varepsilon} \right\rfloor - 1$, this implies that there are at most $\left\lfloor \frac{4\beta}{\varepsilon} \right\rfloor$ leaves in T_j. Then,

$$C_j - a_j = \sum_{k \in T_j} p_k < p_j + \left\lfloor \frac{4\beta}{\varepsilon} \right\rfloor \cdot \frac{\varepsilon}{4} p_j \leq p_j + \beta p_j,$$

where we used $p_k < \frac{\varepsilon}{4} p_j$ for each leaf $k \in T_j$. This contradicts $C_j - a_j \geq (\beta + 1) p_j$.

Ad (C3) Let k be a finished job of height one and let ℓ be the last completing child of k. The parent $\pi(k)$ of k exists because the height of T_j is at least two by (C2) and k is of height one. We create a new instance by replacing ℓ by a job ℓ' with release date $r_{\ell'} := C_k - p_\ell$ and identical processing time, i.e., $p_{\ell'} := p_\ell$. The deadline of ℓ' is $d_{\ell'} := r_{\ell'} + (1 + \varepsilon) p_{\ell'}$.

We argue that k finishes at $r_{\ell'}$ in the new instance and that ℓ' finishes at C_k. Since ℓ is not interrupted, $r_{\ell'} - a_\ell = C_k - p_\ell - a_\ell = C_k - C_\ell$, which is the remaining processing time of k at a_ℓ. If we can show that k is not preempted in $[a_\ell, r_{\ell'})$ in the new instance, k completes at $a_\ell + C_k - C_\ell = r_{\ell'}$. Since ℓ is the last child of k, any job k' released within $[a_\ell, C_k)$ is scheduled later than C_k. (Recall that, after a_j, we restrict to jobs in the interruption tree T_j.) Thus, $p_{k'} \geq \frac{\varepsilon}{4} p_k > p_\ell$. Hence, k is not interrupted in $[a_\ell, r_{\ell'})$ and completes at $r_{\ell'} < C_k \leq d_k$. At time $r_{\ell'}$, job ℓ' is the smallest available job and satisfies $p_{\ell'} < \left(\frac{\varepsilon}{4}\right) p_k < (\frac{\varepsilon}{4})^2 p_{\pi(k)}$. Thus, ℓ' is admitted at $r_{\ell'}$ and is not interrupted until $r_{\ell'} + p_\ell = C_k$ by a similar argumentation about the jobs k' that are released in $[a_\ell, C_k)$. Hence, ℓ' completes at $r_{\ell'} + p_{\ell'} < d_{\ell'}$. Moreover, outside the interval $[a_\ell, C_k)$ neither the instance nor the schedule changed. Since ℓ' is released after k completes, ℓ' becomes a child of $\pi(k)$. This modification does not alter the length of $[a_j, C_j)$ or the number of finished and unfinished jobs. However, k now has one less child. Iteratively applying this modification to any child of k yields that k is now a finished job of height zero. Modifying each finished job of height one proves the claim.

Ad (P1) We prove that no child of $j^\star$ is completely scheduled in $[d_{j^\star}, C_{j^\star})$. If there is a child c with $d_{j^\star} \leq a_c$, it does not prevent the algorithm from finishing $j^\star$ on time. Hence, it could become a child of $\pi(j^\star)$ in the same way we handled the last child of an finished job in the previous claim. That is, we can create a new instance in which c is a child of $\pi(j^\star)$ and $j^\star$ is still unfinished. (See red job in Fig. 5.3.)

Ad (P2) We show that the processing intervals of the children of $j^\star$ form an interval with endpoint $\max\{d_{j^\star}, C_{\max}\}$ where $C_{\max} := \max_{c \in T_{-j^\star}} C_c$. We further prove that they are released and admitted in increasing order of their processing times. More formally, we index the children

in increasing order of their processing times, i.e, $p_{c_1} \leq p_{c_2} \leq \ldots \leq p_{c_t}$. Then, we create a new instance with modified release dates such that each child is released upon completion of the previous child. That is, $r'_{c_t} := \max\{d_{j^\star}, C_{\max}\} - p_{c_t}$ and $r'_{c_{h-1}} := r'_{c_h} - p_{c_{h-1}}$ for $h \in \{2, \ldots, t\}$ where the processing times are not changed, i.e., $p'_{c_h} = p_{c_h}$. In order to ensure that the modified instance is still feasible, we adapt the deadlines $d'_{c_h} := r'_{c_h} + (1+\varepsilon)p'_{c_h}$.

It is left to show that the modifications did not affect the number of finished or unfinished jobs. Obviously, the threshold algorithm still admits every job in $T'_{j^\star}$. A job $k \notin T'_{-j^\star}$ released in $[a_{j^\star}, C_{j^\star})$ satisfies $p_k \geq \frac{\varepsilon}{4} p_{j^\star} > \frac{\varepsilon}{4} p_c$ for all $c \in T_{-j^\star}$. Hence, these jobs do not interrupt either $j^\star$ or any of its children. They are still scheduled after $C_{j^\star}$, and every child $c \in T'_{-j^\star}$ completes before its deadline. We also need to prove that $j^\star$ still cannot finish on time.

If $C_{\max} \leq d_{j^\star}$, every child is completely processed in $[a_{j^\star}, d_{j^\star})$. Hence, job $j^\star$ is still interrupted for the same amount of time before $d_{j^\star}$ in $\mathcal{I}'$ as it is in $\mathcal{I}$. Thus,

$$C'_{j^\star} = a_{j^\star} + p_{j^\star} + \sum_{t \in T'_{-j^\star}} p_c = a_{j^\star} + p_{j^\star} + \sum_{c \in T_{-j^\star}} p_c = C_{j^\star} > d_{j^\star}.$$

If $C_{\max} > d_{j^\star}$, let ℓ be the child in $\mathcal{I}$ with $C_\ell = C_{\max}$. Then, $r_{c'_t} = C_{\max} - p'_{c_t} \leq C_\ell - p_\ell < d_{j^\star}$, where we used that no child is completely processed in $[d_{j^\star}, C_{j^\star})$ by (P1) and that c_t is the child of $j^\star$ with the largest processing time. Thus, the delay of $j^\star$ in $[a_{j^\star}, d_{j^\star})$ is identical to $\sum_{c \in T_{-j^\star}} p_c - (C_\ell - d_{j^\star})$. Hence, $j^\star$ still cannot finish on time. In this case, $C'_{j^\star} = C_{j^\star}$ holds as well. Hence, the modified jobs in $\mathcal{I}'$ still cover the same interval $[a_{j^\star}, C_{j^\star})$. □

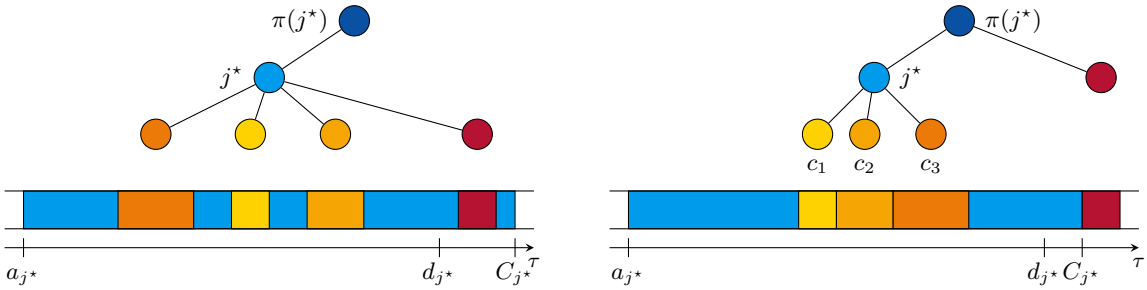

Figure 5.3: Modifications to obtain instance $\mathcal{I}'$ in the proof of Lemma 5.7

Proof of Theorem 5.3

Proof of Theorem 5.3. Let U be the set of jobs that are unfinished by their deadline but whose ancestors (except the machine jobs M_i) have all completed on time. Every job $j \in U$ was admitted by the algorithm at some time a_j with $d_j - a_j \geq \left(1 + \frac{\varepsilon}{2}\right) p_j$. Since j is unfinished, we have $C_j - a_j > d_j - a_j \geq \left(1 + \frac{\varepsilon}{2}\right) p_j$. By Lemma 5.6, $|F_j| - |U_j| \geq \left\lfloor \frac{4 \cdot \varepsilon/2}{\varepsilon} \right\rfloor = 2$. Thus,

$$|T_j| = |F_j| + |U_j| \leq 2|F_j| - 2 < 2|F_j|.$$

Since every ancestor of such a job j finishes on time, this completes the proof. □

5.4 Competitiveness: Admitting Sufficiently Many Jobs

This section shows that the threshold algorithm admits sufficiently many jobs to be $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$-competitive. As mentioned before, this proof is based on the observation that, at time τ, the threshold algorithm admits any available job if its processing time is less than $\frac{\varepsilon}{4}\max_i p_{j_i}$ where p_{j_i} is the job processed by machine i at time τ. We start by formalizing this observation for a class of non-migratory online algorithms before proving that this enables us to bound the number of jobs any feasible schedule successfully schedules during a particular period. Then, we use it to show that the threshold algorithm is indeed $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$-competitive.

5.4.1 A Class of Online Algorithms

In this section, we investigate a class of non-migratory online algorithms. To this end, we generalize the notion of an *available* job as follows: Let $\delta \in (0, \varepsilon)$. We say a job j is available at time τ if it is released before or at time τ, $d_j - \tau \geq (1+\delta)p_j$, and is not yet admitted by the online algorithm.

We consider a non-migratory online algorithm $\mathcal{A}$ satisfying the following properties.

(P1) $\mathcal{A}$ only admits available jobs.

(P2) Retrospectively, for each time τ and each machine i, there is a threshold $u_{i,\tau} \in [0, \infty]$ such that any job j that was available and not admitted by $\mathcal{A}$ at time τ satisfies $p_j \geq u_{i,\tau}$ for every i.

(P3) The function $u^{(i)} : \mathbb{R} \to [0, \infty], \tau \mapsto u_{i,\tau}$ is piece-wise constant and right-continuous for every machine $i \in [m]$. Further, there are only countably many points of discontinuity. (This last property is used to simplify the exposition.)

Key Lemma on the Size of Non-Admitted Jobs

For the proof of the main result in this section, we rely on the following strong, structural lemma about the volume processed by a feasible non-migratory schedule in some time interval and the size of jobs admitted by a non-migratory online algorithm satisfying (P1) and (P2) in the same time interval. We define $u_\tau = \max_i u_{i,\tau}$ for each time point τ.

Let σ be a feasible non-migratory schedule. Without loss of generality, we assume that σ completes all jobs that it started on time. Let X^σ be the set of jobs completed by σ and not admitted by $\mathcal{A}$. For $1 \leq i \leq m$, let X_i^σ be the set of jobs in X^σ processed by machine i. Let C_x be the completion time of job $x \in X^\sigma$ in σ.

Lemma 5.8. *Let $0 \leq \zeta_1 \leq \zeta_2$ and fix $x \in X_i^\sigma$ as well as $Y \subset X_i^\sigma \setminus \{x\}$. If*

(R) $r_x \geq \zeta_1$ *as well as* $r_y \geq \zeta_1$ *for all* $y \in Y$*,*

(C) $C_x \geq C_y$ *for all* $y \in Y$*, and*

(P) $\sum_{y \in Y} p_y \geq \frac{\varepsilon}{\varepsilon - \delta}(\zeta_2 - \zeta_1)$

hold, then $p_x \geq u_{\zeta_2}$*, where* $u_{\zeta_2} = \max_i u_{i,\zeta_2}$ *is the threshold imposed by* $\mathcal{A}$ *at time* ζ_2*. In particular, if* $u_{\zeta_2} = \infty$*, then no such job* x *exists.*

Proof. We show the lemma by contradiction. More precisely, we show that, if $p_x < u_{\zeta_2}$, the schedule σ cannot complete x on time and, hence, is not feasible.

Remember that $x \in X_i^\sigma$ implies that $\mathcal{A}$ did not admit job x at any point τ. At time ζ_2, there are two possible reasons why x was not admitted: $p_x \geq u_{\zeta_2}$ or $d_x - \zeta_2 < (1+\delta)p_x$. In case of the former, the statement of the lemma holds. Toward a contradiction, suppose $p_x < u_{\zeta_2}$ and, thus, $d_x - \zeta_2 < (1+\delta)p_x$ has to hold. As job x arrives with a slack of at least εp_x at its release date r_x and $r_x \geq \zeta_1$ by assumption, we have

$$\zeta_2 - \zeta_1 \geq \zeta_2 - d_x + d_x - r_x > -(1+\delta)p_x + (1+\varepsilon)p_x = (\varepsilon - \delta)p_x. \tag{5.1}$$

Since all jobs in Y complete earlier than x by Assumption (C) and are only released after ζ_1 by (R), the volume processed by σ in $[\zeta_1, C_x)$ on machine i is at least $\frac{\varepsilon}{\varepsilon-\delta}(\zeta_2 - \zeta_1) + p_x$ by (P). Moreover, σ can process at most a volume of $(\zeta_2 - \zeta_1)$ on machine i in $[\zeta_1, \zeta_2)$. These two bounds imply that σ has to process job parts with a processing volume of at least

$$\frac{\varepsilon}{\varepsilon - \delta}(\zeta_2 - \zeta_1) + p_x - (\zeta_2 - \zeta_1) > \frac{\delta}{\varepsilon - \delta}(\varepsilon - \delta)p_x + p_x = (1+\delta)p_x$$

in $[\zeta_2, C_x)$, where the inequality follows using Inequality (5.1). Thus, $C_x > \zeta_2 + (1+\delta)p_x > d_x$, which contradicts the feasibility of σ.

Observe that, by (P1) and (P2), the online algorithm $\mathcal{A}$ admits an available job that satisfies $p_j < u_\tau$. In particular, if $u_\tau = \infty$ for some time point τ, then $\mathcal{A}$ admits any available job. Hence, for $0 \leq \zeta_1 \leq \zeta_2$ with $u_{\zeta_2} = \infty$, there does not exist a job $x \in X_i^\sigma$ and a set $Y \subset X_i^\sigma \setminus \{x\}$ satisfying (R), (C), and (P) for any machine i. □

Bounding the Number of Non-Admitted Jobs

In this section, we use the Properties (P1), (P2), and (P3) to bound the throughput of a non-migratory optimal (offline) algorithm. To this end, we fix an instance as well as an optimal schedule with job set Opt. Let $\mathcal{A}$ be a non-migratory online algorithm satisfying (P1) to (P3).

Let X be the set of jobs in Opt that the algorithm $\mathcal{A}$ did not admit. We assume without loss of generality that all jobs in Opt complete on time. Let $\overline{X} \subseteq X$ be the set of jobs scheduled on any fixed machine with *highest* throughput, i.e., no machine in the optimal schedule processes more jobs from X than $|\overline{X}|$. Without loss of generality, let 1 be a machine where $\mathcal{A}$ achieves

lowest throughput. Assumption (P3) guarantees that the threshold $u_{1,\tau}$ is piece-wise constant and right-continuous, i.e., $u^{(1)}$ is constant on intervals of the form $[\tau_t, \tau_{t+1})$. Let $\underline{I}$ represent the set of maximal intervals $I_t = [\tau_t, \tau_{t+1})$ where $u^{(1)}$ is constant. That is, $u_{1,\tau} = \underline{u_t}$ holds for all $\tau \in I_t$ and $u_{1,\tau_{t+1}} \neq \underline{u_t}$, where $\underline{u_t} := u_{1,\tau_t}$, The main result of this section is the following theorem.

Theorem 5.9. *Let $\overline{X}$ be the set of jobs that are scheduled on a machine with highest throughput in an optimal schedule. Let $\underline{I} = \{I_1, \dots, I_T\}$ be the set of maximal intervals on a machine of $\mathcal{A}$ with lowest throughput such that the machine-dependent threshold is constant for each interval and has the value $\underline{u_t}$ in interval $I_t = [\tau_t, \tau_{t+1})$. Then,*

$$|\overline{X}| \leq \sum_{t=1}^{T} \frac{\varepsilon}{\varepsilon - \delta} \frac{\tau_{t+1} - \tau_t}{\underline{u_t}} + T,$$

where we set $\frac{\tau_{t+1}-\tau_t}{\underline{u_t}} = 0$ if $\underline{u_t} = \infty$ and $\frac{\tau_{t+1}-\tau_t}{\underline{u_t}} = \infty$ if $\{\tau_t, \tau_{t+1}\} \cap \{-\infty, \infty\} \neq \emptyset$ and $\underline{u_t} < \infty$.

We observe that $T = \infty$ trivially proves the statement as $\overline{X}$ contains at most finitely many jobs. The same is true if $\frac{\tau_{t+1}-\tau_t}{\underline{u_t}} = \infty$ for some $t \in [T]$. Hence, for the remainder of this section we assume without loss of generality that $\underline{I}$ only contains finitely many intervals and that $\frac{\tau_{t+1}-\tau_t}{\underline{u_t}} < \infty$ holds for every $t \in [T]$.

To prove this theorem, we develop a charging scheme that assigns jobs $x \in \overline{X}$ to intervals in $\underline{I}$. The idea behind our charging scheme is that Opt does not contain arbitrarily many jobs that are available in I_t since $\underline{u_t}$ provides a natural lower bound on their processing times. In particular, the processing time of any job that is *released* during interval I_t and not admitted by the algorithm exceeds the lower bound $\underline{u_t}$. Thus, the charging scheme relies on the release date r_x and the size p_x of a job $x \in \overline{X}$ as well as on the precise structure of the intervals created by $\mathcal{A}$.

The charging scheme we develop is based on a careful modification of the following partition $(F_t)_{t=1}^{T}$ of the set $\overline{X}$. Fix an interval $I_t \in \underline{I}$ and define the set $F_t \subseteq \overline{X}$ as the set that contains all jobs $x \in \overline{X}$ released during I_t, i.e., $F_t = \{x \in \overline{X} : r_x \in I_t\}$. Since, upon release, each job $x \in \overline{X}$ is available and not admitted by $\mathcal{A}$, the next fact directly follows from Properties (P1) and (P2).

Fact 5.10. *For all jobs $x \in F_t$ it holds $p_x \geq \underline{u_t}$. In particular, if $\underline{u_t} = \infty$, then $F_t = \emptyset$.*

In fact, the charging scheme maintains this property and only assigns jobs in $\overline{X}$ to intervals I_t if $p_x \geq \underline{u_t}$. In particular, no job will be assigned to an interval with $\underline{u_t} = \infty$.

We now formalize how many jobs in $\overline{X}$ are assigned to a specific interval I_t. Let

$$\varphi_t := \left\lfloor \frac{\varepsilon}{\varepsilon - \delta} \frac{\tau_{t+1} - \tau_t}{\underline{u_t}} \right\rfloor + 1$$

if $\underline{u_t} < \infty$, and $\varphi_t = 0$ if $\underline{u_t} = \infty$. We refer to $\underline{u_t}$ as the *target number* of I_t. As discussed before, we assume $\frac{\tau_{t+1}-\tau_t}{\underline{u_t}} < \infty$, and, thus, the target number is well-defined. If each of the sets F_t satisfies $|F_t| \leq \varphi_t$, then Theorem 5.9 immediately follows. In general, $|F_t| \leq \varphi_t$ does not have to be true since jobs in Opt may be preempted and processed during several intervals I_t. Therefore, for proving Theorem 5.9, we show that there always exists another partition $(G_t)_{t=1}^T$ of $\overline{X}$ such that $|G_t| \leq \varphi_t$ holds.

The high-level idea of this proof is the following: Consider an interval $I_t = [\tau_t, \tau_{t+1})$. If F_t does not contain too many jobs, i.e., $|F_t| \leq \varphi_t$, we would like to set $G_t = F_t$. Otherwise, we find a later interval $I_{t'}$ with $|F_{t'}| < \varphi_{t'}$ such that we can assign the excess jobs in F_t to $I_{t'}$.

Proof of Theorem 5.9. As observed, it suffices to show the existence of a partition $\mathcal{G} = (G_t)_{t=1}^T$ of $\overline{X}$ such that $|G_t| \leq \varphi_t$ in order to prove the theorem.

In order to repeatedly apply Lemma 5.8, we only assign excess jobs $x \in F_t$ to $G_{t'}$ for $t < t'$ if their processing time is at least the threshold of $I_{t'}$, i.e., $p_x \geq \underline{u_{t'}}$. By our choice of parameters, a set $G_{t'}$ with $\varphi_{t'}$ many jobs of size at least $\underline{u_{t'}}$ "covers" the interval $I_{t'} = [\tau_{t'}, \tau_{t'+1})$ as often as required by (P) in Lemma 5.8, i.e.,

$$\sum_{x \in G_{t'}} p_x \geq \varphi_{t'} \cdot \underline{u_{t'}} = \left(\left\lfloor \frac{\varepsilon}{\varepsilon - \delta} \frac{\tau_{t'+1} - \tau_{t'}}{\underline{u_{t'}}} \right\rfloor + 1 \right) \cdot \underline{u_{t'}} \geq \frac{\varepsilon}{\varepsilon - \delta} (\tau_{t'+1} - \tau_{t'}). \tag{5.2}$$

The proof consists of two parts: the first one is to inductively (on t) construct the partition $\mathcal{G} = (G_t)_{t=1}^T$ of $\overline{X}$, where $|G_t| \leq \varphi_t$ holds for $t \in [T-1]$. The second one is the proof that a job $x \in G_t$ satisfies $p_x \geq \underline{u_t}$ which will imply $|G_T| \leq \varphi_T$. During the construction of $\mathcal{G}$ we define temporary sets $A_t \subset \overline{X}$ for intervals I_t. The set G_t is chosen as a subset of $F_t \cup A_t$ of appropriate size. In order to apply Lemma 5.8 to each job in A_t individually, alongside A_t, we construct a set $Y_{x,t}$ and a time $\tau_{x,t} \leq r_x$ for each job $x \in \overline{X}$ that is added to A_t. Let C_y^* be the completion time of some job $y \in \overline{X}$ in the optimal schedule Opt. The second part of the proof is to show that x, $\tau_{x,t}$, and $Y_{x,t}$ satisfy

(R) $r_y \geq \tau_{x,t}$ for all $y \in Y_{x,t}$,

(C) $C_x^* \geq C_y^*$ for all $y \in Y_{x,t}$, and

(P) $\sum_{y \in Y_{x,t}} p_y \geq \frac{\varepsilon}{\varepsilon - \delta}(\tau_t - \tau_{x,t})$.

This implies that x, $Y = Y_{x,t}$, $\zeta_1 = \tau_{x,t}$, and $\zeta_2 = \tau_t$ satisfy the conditions of Lemma 5.8, and thus the processing time of x is at least the threshold at time τ_t, i.e., $p_x \geq u_{\tau_t} \geq \underline{u_t}$.

Constructing $\mathcal{G} = (G_t)_{t=1}^T$. We inductively construct the sets G_t in the order of their indices. We start by setting $A_t = \emptyset$ for all intervals I_t with $t \in T$. We define $Y_{x,t} = \emptyset$ for each job $x \in \overline{X}$ and each interval I_t. The preliminary value of the time $\tau_{x,t}$ is the minimum of the

starting point τ_t of the interval I_t and the release date r_x of x, i.e., $\tau_{x,t} := \min\{\tau_t, r_x\}$. We refer to the step in the construction where G_t was defined by *step* t.

Starting with $t = 1$, let I_t be the next interval to consider during the construction with $t < T$. Depending on the cardinality of $F_t \cup A_t$, we distinguish two cases. If $|F_t \cup A_t| \leq \varphi_t$, then we set $G_t = F_t \cup A_t$.

If $|F_t \cup A_t| > \varphi_t$, then we order the jobs in $F_t \cup A_t$ in increasing order of completion times in the optimal schedule. The first φ_t jobs are assigned to G_t while the remaining $|F_t \cup A_t| - \varphi_t$ jobs are added to A_{t+1}. In this case, we might have to redefine the times $\tau_{x,t+1}$ and the sets $Y_{x,t+1}$ for the jobs x that were newly added to A_{t+1}. Fix such a job x. If there is no job z in the just defined set G_t that has a smaller release date than $\tau_{x,t}$, we set $\tau_{x,t+1} = \tau_{x,t}$ and $Y_{x,t+1} = Y_{x,t} \cup G_t$. Otherwise let $z \in G_t$ be a job with $r_z < \tau_{x,t}$ that has the smallest time $\tau_{z,t}$. We set $\tau_{x,t+1} = \tau_{z,t}$ and $Y_{x,t+1} = Y_{z,t} \cup G_t$.

Finally, we set $G_T = F_T \cup A_T$. We observe that $\underline{u_T} < \infty$ implies $\varphi_T = \infty$ because $\tau_{T+1} = \infty$. Since this contradicts the assumption $\varphi_t < \infty$ for all $t \in [T]$, this implies $\underline{u_T} = \infty$. We will show that $p_x \geq \underline{u_T}$ for all $x \in G_T$. Hence, $G_T = \emptyset$. Therefore $|G_T| = \varphi_T = 0$.

Bounding the size of the jobs in G_t. We consider the intervals again in increasing order of their indices and show by induction that any job x in G_t satisfies $p_x \geq \underline{u_t}$ which implies $G_t = \emptyset$ if $\underline{u_t} = \infty$. Clearly, if $x \in F_t \cap G_t$, Fact 5.10 guarantees $p_x \geq \underline{u_t}$. Hence, in order to show the lower bound on the processing time of $x \in G_t$, it is sufficient to consider jobs in $G_t \setminus F_t \subset A_t$. To this end, we show that for such jobs (R), (C), and (P) are satisfied. Thus, Lemma 5.8 guarantees that $p_x \geq u_{\tau_t} \geq \underline{u_t}$ because $u_{\tau_t} \geq u_{1,\tau_t} = \underline{u_t}$ by definition. Hence, $A_t = \emptyset$ if $\underline{u_t} = \infty$ since in this case the global bound is also unbounded, i.e., $u_{\tau_t} \geq \underline{u_t} = \infty$.

By construction, $A_1 = \emptyset$. Hence, (R), (C), and (P) are satisfied for each job $x \in A_1$.

Suppose that the Conditions (R), (C), and (P) are satisfied for all $x \in A_s$ for all $1 \leq s < t$. Hence, for $s < t$, the set G_s only contains jobs x with $p_x \geq \underline{u_s}$. Fix $x \in A_t$. We want to show that $p_x \geq \underline{u_t}$. By the induction hypothesis and by Fact 5.10, $p_y \geq \underline{u_{t-1}}$ holds for all $y \in G_{t-1}$. Since x did not fit in G_{t-1} anymore, $|G_{t-1}| = \varphi_{t-1}$.

We distinguish two cases based on G_{t-1}. If there is no job $z \in G_{t-1}$ with $r_z < \tau_{x,t-1}$, then $\tau_{x,t} = \tau_{x,t-1}$, and (R) and (C) are satisfied by construction and by the induction hypothesis. For (P), consider

$$\begin{aligned}
\sum_{y \in Y_{x,t}} p_y &= \sum_{y \in Y_{x,t-1}} p_y + \sum_{y \in G_{t-1}} p_y \\
&\geq \frac{\varepsilon}{\varepsilon - \delta}(\tau_{t-1} - \tau_{x,t-1}) + \underline{u_{t-1}} \cdot \varphi_{t-1} \\
&\geq \frac{\varepsilon}{\varepsilon - \delta}(\tau_{t-1} - \tau_{x,t-1}) + \frac{\varepsilon}{\varepsilon - \delta}(\tau_t - \tau_{t-1}) \\
&= \frac{\varepsilon}{\varepsilon - \delta}(\tau_t - \tau_{x,t}) \,,
\end{aligned}$$

where the first inequality holds due to the induction hypothesis. By Lemma 5.8, $p_x \geq u_{\tau_t} \geq \underline{u_t}$.

If there is a job $z \in G_{t-1}$ with $r_z < \tau_{x,t-1} \leq \tau_{t-1}$, then $z \in A_{t-1}$. In step $t-1$, we chose z with minimal $\tau_{z,t-1}$. Thus, $r_y \geq \tau_{y,t-1} \geq \tau_{z,t-1}$ for all $y \in G_{t-1}$ and $r_x \geq \tau_{x,t-1} > r_z \geq \tau_{z,t-1}$ which is Condition (R) for the jobs in G_{t-1}. Moreover, by the induction hypothesis, $r_y \geq \tau_{z,t-1}$ holds for all $y \in Y_{z,t-1}$. Thus, $\tau_{x,t}$ and $Y_{x,t}$ satisfy (R). For (C), consider that $C^*_x \geq C^*_y$ for all $y \in G_{t-1}$ by construction and, thus, $C^*_x \geq C^*_z \geq C^*_y$ also holds for all $y \in Y_{z,t-1}$ due to the induction hypothesis. For (P), observe that

$$\begin{aligned}\sum_{y \in Y_{x,t}} p_y &= \sum_{y \in Y_{z,t-1}} p_y + \sum_{y \in G_{t-1}} p_y \\ &\geq \frac{\varepsilon}{\varepsilon - \delta}(\tau_{t-1} - \tau_{z,t-1}) + \underline{u_{t-1}} \cdot \varphi_{t-1} \\ &\geq \frac{\varepsilon}{\varepsilon - \delta}(\tau_{t-1} - \tau_{z,t-1}) + \frac{\varepsilon}{\varepsilon - \delta}(\tau_t - \tau_{t-1}) \\ &\geq \frac{\varepsilon}{\varepsilon - \delta}(\tau_t - \tau_{x,t}).\end{aligned}$$

Here, the first inequality follows from the induction hypothesis and the second from the definition of $\underline{u_{t-1}}$ and φ_{t-1}. Hence, Lemma 5.8 implies $p_x \geq u_{\tau_t} \geq \underline{u_t}$.

We note that $p_x \geq \underline{u_t}$ for all $x \in G_t$ and for all $t \in [T]$.

Bounding $|\overline{X}|$. By construction, we know that $\bigcup_{t=1}^{T} G_t = \overline{X}$. We start with considering intervals I_t with $\underline{u_t} = \infty$. Then, I_t has an unbounded threshold, i.e., $u_\tau = \infty$ for all $\tau \in I_t$, and $F_t = \emptyset$ by Fact 5.10. In the previous part we have seen that the conditions for Lemma 5.8 are satisfied. Hence, $G_t = \emptyset$ if $\underline{u_t} = \infty$. For t with $\underline{u_t} < \infty$, we have $|G_t| \leq \varphi_t = \left\lfloor \frac{\varepsilon}{\varepsilon - \delta} \frac{\tau_{t+1} - \tau_t}{\underline{u_t}} \right\rfloor + 1$. As explained before, this bounds the number of jobs in $\overline{X}$. □

5.4.2 Admitting Sufficiently Many Jobs

In this section, we show the following theorem and give the proof of Theorem 5.1.

Theorem 5.11. *An optimal non-migratory (offline) algorithm completes at most a factor* $\left(\frac{8}{\varepsilon} + 4\right)$ *more jobs on time than admitted by the threshold algorithm.*

Proof. As in the previous section, fix an instance and an optimal solution Opt. Let X be the set of jobs in Opt that the threshold algorithm did not admit. We assume without loss of generality that all jobs in Opt finish on time. Further, let J denote the set of jobs that the threshold algorithm admitted. Then, $X \cup J$ is a superset of the jobs in Opt. Thus, $|X| \leq \left(\frac{8}{\varepsilon} + 3\right)|J|$ implies Theorem 5.11.

To this end, let $\overline{X} \subseteq X$ denote the jobs in Opt scheduled on a machine with highest throughput. Without loss of generality, let 1 be again a machine where the threshold algorithm achieves lowest throughput. Let $\underline{J}$ denote the jobs scheduled by the threshold algorithm on the first machine. Then, showing $|\overline{X}| \leq \left(\frac{8}{\varepsilon} + 3\right)|\underline{J}|$ suffices to prove the main result of this section.

Given that the threshold algorithm satisfies Assumptions (P1), (P2), and (P3), Theorem 5.9 already provides a bound on the cardinality of $\overline{X}$ in terms of the *intervals* corresponding to the schedule on the least loaded machine. Thus, it remains to show that the threshold algorithm indeed qualifies for applying Theorem 5.9 and that the bound developed therein can be translated to a bound in terms of $|\underline{J}|$.

We start by showing that the threshold algorithm satisfies the assumptions necessary for applying Theorem 5.9. Clearly, as the threshold algorithm only admits a job j at time τ if $d_j - \tau \geq \left(1 + \frac{\varepsilon}{2}\right)p_j$, setting $\delta = \frac{\varepsilon}{2}$ proves that the threshold algorithm satisfies (P1). For (P2), we retrospectively analyze the schedule generated by the threshold algorithm. For a time τ, let j_i denote the job scheduled on machine i. Then, setting $u_{i,\tau} := \frac{\varepsilon}{4}p_{j_i}$ or $u_{i,\tau} = \infty$ if no such job j_i exists, indeed provides us with the machine-dependent threshold necessary for (P2). This discussion also implies that $u^{(i)}$ has only countably many points of discontinuity as there are only finitely many jobs in the instance, and that $u^{(i)}$ is right-continuous.

Hence, let $\underline{I}$ denote the set of maximal intervals $I_t = [\tau_t, \tau_{t+1})$ for $t \in [T]$ of constant threshold $u_{1,\tau}$. Thus, by Theorem 5.9,

$$|\overline{X}| \leq \sum_{t=1}^{T} \frac{\varepsilon}{\varepsilon - \delta} \frac{\tau_{t+1} - \tau_t}{\underline{u_t}} + T. \tag{5.3}$$

As the threshold $u_{i,\tau}$ is proportional to the processing time of the job currently scheduled on machine i, the interval I_t either represents an idle interval of machine 1 (with $u_{1,\tau} = \infty$) or corresponds to the uninterrupted processing of some job j on machine 1. We denote this job by j_t if it exists. We consider now the set $\underline{I}_j \subseteq \underline{I}$ of intervals with $j_t = j$ for some particular job $j \in \underline{J}$. As observed, these intervals correspond to job j being processed which happens for a total of p_j units of time. Combining with $\underline{u_t} = \frac{\varepsilon}{4}p_j$ for $I_t \in \underline{I}_j$, we get

$$\sum_{t:I_t \in \underline{I}_j} \frac{\tau_{t+1} - \tau_t}{\underline{u_t}} = \frac{p_j}{\frac{\varepsilon}{4}p_j} = \frac{4}{\varepsilon}.$$

As $\delta = \frac{\varepsilon}{2}$, we additionally have that $\frac{\varepsilon}{\varepsilon - \delta} = 2$. Hence, we rewrite Equation (5.3) by

$$|\overline{X}| \leq \frac{8}{\varepsilon}|\underline{J}| + T.$$

It remains to bound T in terms of $|\underline{J}|$ to conclude the proof. To this end, we recall that the admission of a job j to a machine interrupts the processing of at most one previously admitted job. Hence, the admission of $|\underline{J}|$ jobs to machine 1 creates at most $2|\underline{J}| + 1$ intervals.

If the threshold algorithm does not admit any job to machine 1 with lowest throughput, i.e., $|\underline{J}| = 0$, then $u_{1,\tau} = \infty$ for each time point τ. Hence, there exists no job in the instance that the threshold algorithm did not admit. Thus, $|X| \leq |\mathcal{J}| = |J|$ which completes the proof.

Otherwise, $2|\underline{J}| + 1 \leq 3|\underline{J}|$. Therefore,

$$|\overline{X}| \leq \left(\frac{8}{\varepsilon} + 3\right) |\underline{J}|.$$

Combining with the observation about $\overline{X}$ and $\underline{J}$ previously discussed, we obtain

$$|\textsc{Opt}| \leq |X \cup J| \leq m|\overline{X}| + |J| \leq m\left(\frac{8}{\varepsilon} + 3\right)|\underline{J}| + |J| \leq \left(\frac{8}{\varepsilon} + 4\right)|J|,$$

which concludes the proof. □

Finalizing the proof of Theorem 5.1

Proof of Theorem 5.1. In Theorem 5.3 we show that the threshold algorithm completes at least half of all admitted jobs J on times. Theorem 1.1 in [KP01] (Theorem 5.2) gives a bound on the throughput of an optimal migratory schedule in terms of the throughput of an optimal non-migratory solution. In Theorem 5.9, we bound the throughput $|\textsc{Opt}|$ of an optimal non-migratory solution in terms of $|J|$. Combining these theorems shows that the threshold algorithm achieves a competitive ratio of $c = 6 \cdot 2 \cdot \left(\frac{8}{\varepsilon} + 4\right) = \frac{96}{\varepsilon} + 48$. □

5.5 Lower Bound on the Competitive Ratio

We give a lower bound, that (up to constants) matches our upper bound in Theorem 5.1. This shows that the threshold algorithm is best possible for online throughput maximization.

Theorem 5.12. *Every deterministic online algorithm has a competitive ratio* $\Omega\left(\frac{1}{\varepsilon}\right)$.

The proof idea is as follows: We release $\Omega\left(\frac{1}{\varepsilon}\right)$ *levels* of jobs. In each level, the release date of any but the first job is the deadline of the previous job. Whenever an online algorithm decides to complete a job from level ℓ (provided no further jobs are released), then the release of jobs in level ℓ stops and a sequence of $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$ jobs in level $\ell + 1$ is released. Jobs in level $\ell + 1$ have processing time that is too large to fit in the slack of a job of level ℓ. Thus, an algorithm has to discard the job started at level ℓ to run a job of level $\ell + 1$. This implies that it can only finish one job while the optimum can finish a job from every other level.

Proof of Theorem 5.12. Let $\varepsilon < \frac{1}{10}$ such that $\frac{1}{8\varepsilon} \in \mathbb{N}$. Toward a contradiction, suppose there is an online algorithm with competitive ratio $c < \frac{1}{8\varepsilon}$. We construct an adversarial instance in which each job j belongs to one of $2 \cdot \lceil c + 1 \rceil$ levels and fulfills $d_j = r_j + (1 + \varepsilon) \cdot p_j$. The processing time for any job j in level ℓ is $p_j = p^{(\ell)} = (2\varepsilon)^\ell$. This (along with the interval structure) makes sure that no two jobs from consecutive levels can both be completed by a single schedule, which we will use to show that the online algorithm can only complete a single job throughout the entire instance. The decrease in processing times between levels, however,

makes sure that the optimum finishes a job from every other level, resulting in an objective value of $\lceil c+1 \rceil$, which contradicts the algorithm being c-competitive.

The sequence starts with level 0 at time 0 with the release of one job j with processing time $p^{(0)} = 1$ and, thus, deadline $d_j = 1 + \varepsilon$. We will show inductively that, for each level ℓ, there is a time t_ℓ when there is only a single job j_ℓ left that the algorithm can still finish, and this job is from the current level ℓ and, thus, $p_{j_\ell} = p^{(\ell)} = (2\varepsilon)^\ell$. We will also make sure that at t_ℓ at most a $\left(\frac{2}{3}\right)$-fraction of the time window of j_ℓ has passed. From t_ℓ on, no further jobs from level ℓ are released, and jobs from level $\ell+1$ start being released or, if $\ell = 2 \cdot \lceil c+1 \rceil - 1$, we stop releasing jobs altogether. It is clear that t_0 exists.

Consider some time t_ℓ, and we will release jobs from level $\ell+1$ leading to time $t_{\ell+1}$. The first job j from level $\ell+1$ has release date t_ℓ and, by the above constraints, $d_j = t_\ell + (1+\varepsilon) \cdot p_j$, where $p_j = p^{(\ell+1)} = (2\varepsilon)^{\ell+1}$. As long as no situation occurs that fits the above description of $t_{\ell+1}$, we release an additional job of level $\ell+1$ at the deadline of the previous job from this level (with identical time-window length and processing time). We show that we can find time $t_{\ell+1}$ before $\frac{1}{8\varepsilon}$ jobs from level $\ell+1$ have been released. Note that the deadline of the $\frac{1}{8\varepsilon}$th job from level $\ell+1$ is $t_\ell + \frac{1}{8\varepsilon} \cdot (1+\varepsilon) \cdot 2\varepsilon \cdot p^{(\ell)}$, which is smaller than the deadline of d_{j_ℓ} since $d_{j_\ell} - t_\ell \geq \frac{1}{3} \cdot p^{(\ell)}$ by the induction hypothesis and $\varepsilon < \frac{1}{10}$. This shows that, unless more than $\frac{1}{8\varepsilon}$ jobs from level $\ell+1$ are released (which will not happen as we will show), all time windows of jobs from level $\ell+1$ are contained in that of j_ℓ.

Note that there must be a job $j^\star$ among the $\frac{1}{8\varepsilon}$ first ones in level $\ell+1$ that the algorithm completes if no further jobs are released within the time window of $j^\star$: By the induction hypothesis, the algorithm can only hope to finish a single job released before time t_ℓ and the optimum could complete $\frac{1}{8\varepsilon}$ jobs from level $\ell+1$, so $j^\star$ must exist for the algorithm to be c-competitive. Now we can define $j_{\ell+1}$ to be the first such job $j^\star$ and find $t_{\ell+1}$ within its time window: At the release date of $j^\star$, the algorithm could only complete j_ℓ. However, since the algorithm finishes $j_{\ell+1}$ if there are no further jobs released, and $\varepsilon < \frac{1}{10}$, it must have worked on $j_{\ell+1}$ for more than $\frac{p^{(\ell+1)}}{2}$ units of time until $r_{\ell+1} + \frac{2}{3} \cdot p^{(\ell+1)} =: t_{\ell+1}$. This quantity, however, exceeds the slack of j_ℓ, meaning that the algorithm cannot finish j_ℓ anymore as the slack of j_ℓ is $\varepsilon p^{(\ell)} = 2^\ell \varepsilon^{\ell+1}$. Therefore, $t_{\ell+1}$ has the desired properties.

This defines $t_{2 \cdot \lceil c+1 \rceil}$, and indeed the algorithm will only finish a single job. We verify that an optimal algorithm can schedule a job from every other level. Note that, among levels of either parity, processing times are decreasing by a factor of $4\varepsilon^2$ between consecutive levels. So, for any job j, the total processing time of jobs other than j that need to be processed within the time window of j adds up to less than

$$\begin{aligned}\sum_{\ell=1}^{\infty} (4\varepsilon^2)^\ell \cdot p_j = 4\varepsilon^2 \cdot \sum_{\ell=0}^{\infty} (4\varepsilon^2)^\ell \cdot p_j &= \frac{4\varepsilon^2}{1-4\varepsilon^2} \cdot p_j \\ &\leq \frac{4}{10} \cdot \frac{1}{1-\frac{4}{100}} \cdot \varepsilon p_j < \varepsilon \cdot p_j = d_j - r_j - p_j,\end{aligned}$$

which completes the proof. □

5.6 Concluding Remarks

We provide an online algorithm for scheduling deadline-sensitive jobs on identical parallel machines. We close the problem with the best (up to constants) competitive ratio $\Theta\left(\frac{1}{\varepsilon}\right)$.

Our lower bound points at two research directions: First, it is constructed on a single machine and it is not immediately clear how to translate this to the multiple-machine setting. In fact, the impossibility result for jobs without slack also relies on a single machine and, up to date, it is not yet answered if slack is even necessary for achieving non-trivial competitive ratios in the presence of multiple machines. Moreover, for the more tractable problem of machine utilization the competitive ratio even improves with an increasing number of machines as shown in [SS16].

Second, we only use unit-weight jobs in the lower bound as this is the setting we are mostly interested in. However, there is no better lower bound in the weighted setting. That is, there is still a gap between our lower bound $\Omega\left(\frac{1}{\varepsilon}\right)$ and the upper bound $\mathcal{O}\left(\frac{1}{\varepsilon^2}\right)$ by Lucier et al. [LMNY13]. It would be interesting to close this gap. The analysis of our algorithm crucially relies on the fact that jobs are only preempted by significantly smaller jobs. In the weighted variant, interruption must also happen for longer yet more valuable jobs, which shows that one would need to develop new techniques to improve the analysis. Of course, it is also possible that there is another algorithm with yet another analysis that closes this gap.

Another interesting question asks whether randomization allows for improved results. On a single machine, there is indeed an $\mathcal{O}(1)$-competitive randomized algorithm, even without any slack assumption [KP03]. We are not aware of lower bounds that rule out similar results on multiple machines.

6

Online Throughput Maximization with Commitment

We consider again online throughput maximization where jobs with deadlines arrive online over time at their release dates. The task is to find a preemptive schedule on m machines maximizing the number of jobs that finish on time. We quantify the impact that provider commitment requirements have on the performance of online algorithms. We require again that jobs contain some *slack* $\varepsilon > 0$. We present the first online algorithm for handling commitment on parallel machines for arbitrary slack ε. When the scheduler must commit upon starting a job, the algorithm is $\Theta\left(\frac{1}{\varepsilon}\right)$-competitive. Somewhat surprisingly, this is the same optimal performance bound (up to constants) as for scheduling without commitment. If commitment decisions must be made before a job's slack becomes less than a δ-fraction of its processing time, we prove a competitive ratio of $\mathcal{O}\left(\frac{1}{\varepsilon-\delta}\right)$ for $0 < \delta < \varepsilon$. This result nicely interpolates between commitment upon starting a job and commitment upon arrival. For the latter model, we show that no (randomized) online algorithm admits a bounded competitive ratio.
Finally, we observe that for scheduling with commitment restricting to unit weights is essential; for job-dependent weights, we rule out competitive deterministic algorithms.
Bibliographic Remark: The presented lower bounds are based on joint work with L. Chen, N. Megow, K. Schewior, and C. Stein [CEM+20]. The algorithm and its analysis are based on joint work with N. Megow and K. Schewior [EMS20]. Therefore, some parts correspond to or are identical with [CEM+20] and [EMS20].

Table of Contents

6.1 Introduction

The model we consider in this chapter is almost identical to the one in Chapter 5. To recap, jobs from an unknown job set $\mathcal{J}$ arrive online over time at their *release dates* r_j. Each job $j \in \mathcal{J}$ has a *processing time* $p_j \geq 0$ and a *deadline* d_j. There are m identical parallel machines to process these jobs or a subset of them. A job is said to *complete* if it receives p_j units of processing time within the interval $[r_j, d_j)$. We allow *preemption*, i.e., the processing of a job can be interrupted at any time. We distinguish schedules *with* and *without migration*. If we allow migration, then a preempted job can resume processing on any machine whereas it must run completely on the same machine otherwise. The task is to find a feasible schedule with maximum throughput. In the three-field notation by Graham et al. [GLLRK79b], this problem is denoted by $P \,|\, \text{online } r_j, \text{pmtn} \,|\, \sum(1 - U_j)$.

We assess the performance of online algorithms with standard *competitive analysis*. This means, we compare the throughput of an online algorithm with the throughput achievable by an optimal offline algorithm that knows the job set in advance. To circumvent known lower bounds involving "tight" jobs with $d_j - r_j \approx p_j$, we require that jobs contain some *slack* $\varepsilon > 0$, i.e., every job j satisfies $d_j - r_j \geq (1 + \varepsilon)p_j$. As in the previous chapter, the competitive ratio of our online algorithm will be a function of ε; the greater the slack, the better should the performance of our algorithm be.

In contrast to Chapter 5, we focus on the question how to handle *commitment* requirements in online throughput maximization. Modeling commitment addresses the issue that a high-throughput schedule may abort jobs close to their deadlines in favor of many shorter and more urgent tasks [FBK+12], which may not be acceptable for the job owner. Consider a company that starts outsourcing mission-critical processes to external clouds and that may require a certain provider-side guarantee, i.e., service providers have to *commit to complete* admitted jobs before they cannot be moved to other computing clusters anymore. In other situations, a commitment to complete jobs might be required even earlier just before starting the job, e.g., for a faultless copy of a database as companies tend to rely on business analytics to support decision making. Since analytical tools, which usually work with copies of databases, depend on faultless data, the completion of such a copy process must be guaranteed once it started.

We distinguish three different models for scheduling with commitment: (i) *commitment upon job arrival*, (ii) *commitment upon job admission*, and (iii) *δ-commitment*. In the first, most restrictive model, an algorithm must decide immediately at a job's release date if the job will be completed or not. In the second model, an algorithm may discard a job any time before its start, its admission. This reflects the situation when the start of a process is the critical time point after which the successful execution is essential (e.g., faultless copy of a database). In the third model, δ-commitment, an online algorithm must commit to complete a job when its slack has reduced from the original slack requirement of at least an ε-fraction of the job size to a δ-fraction for $0 < \delta < \varepsilon$. Then, the latest feasible time for committing to job j is $d_j - (1+\delta)p_j$.

This models an early-enough commitment (parameterized by δ) for mission-critical jobs.

Previous results For related work on online throughput maximization without commitment requirements, we refer to the previous chapter and the references therein.

Commitment upon job arrival In the most restrictive model, Lucier et al. [LMNY13] rule out competitive online algorithms for any slack parameter ε when jobs have arbitrary weights.

The special case $w_j = p_j$, or machine utilization, is much more tractable than weighted or unweighted throughput maximization. A simple greedy algorithm achieves the best possible competitive ratio $\frac{1+\varepsilon}{\varepsilon}$ on a single machine, even for commitment upon arrival, as shown by the analysis of DasGupta and Palis [DP00] and the matching lower bound by Garay et al. [GNYZ02]. For scheduling with commitment upon arrival on m parallel identical machines, there is an $\mathcal{O}(\sqrt[m]{1/\varepsilon})$-competitive algorithm and an almost matching lower bound by Schwiegelshohn and Schwiegelshohn [SS16]. When preemption is not allowed, Goldwasser and Kerbikov [GK03] give a best possible $\left(2+\frac{1}{\varepsilon}\right)$-competitive algorithm on a single machine. Very recently, Jamalabadi, Schwiegelshohn, and Schwiegelshohn [JSS20] extend this model to parallel machines; their algorithm is near optimal with a performance guarantee approaching $\ln \frac{1}{\varepsilon}$ as m tends to infinity.

Commitment upon admission and δ-commitment In our previous work [CEM+20], we give a more elaborate variant of the threshold algorithm that achieves the first non-trivial upper bound for both models on a single machine. For *commitment upon job admission*, Lucier et al. [LMNY13] give a heuristic that empirically performs very well but for which they cannot show a rigorous worst-case bound. In fact, later, Azar et al. [AKL+15] show that no bounded competitive ratio is possible for weighted throughput maximization for small ε. For $\delta = \frac{\varepsilon}{2}$ in the *δ-commitment model*, they design (in the context of truthful mechanisms) an online algorithm that is $\Theta\left(\frac{1}{\sqrt[3]{1+\varepsilon}-1} + \frac{1}{(\sqrt[3]{1+\varepsilon}-1)^2}\right)$-competitive if the slack ε is sufficiently large, i.e., if $\varepsilon > 3$. They leave open if this latter condition is an inherent property of any committed scheduler in this model, and our lower bound for weights answers this affirmatively.

Machine utilization is better understood: We note that, as commitment upon arrival clearly is more restrictive than commitment upon admission and δ-commitment, the previously mentioned results immediately carry over and provide bounded competitive ratios. Without preemption, Goldwasser [Gol03] gives an optimal $\left(2+\frac{1}{\varepsilon}\right)$-competitive algorithm on a single machine and Lee [Lee03] gives an $\mathcal{O}\left(\frac{m}{\sqrt[m]{\varepsilon}}\right)$-competitive algorithm on m parallel identical machines.

Our contribution Our main result is an algorithm that is best possible (up to constant factors) for online throughput maximization with commitment on parallel identical machines. Our algorithm does not migrate jobs and still achieves a competitive ratio that matches the

general lower bound for migratory algorithms. Further, we show a strong lower bound for scheduling with commitment upon job arrival, even for randomized algorithms.

Impossibility result for commitment upon job arrival In this most restrictive model, an algorithm must decide immediately at a job's release date if the job will be completed or not. We show that no (randomized) online algorithm admits a bounded competitive ratio. Such a lower bound has only been shown by exploiting arbitrary job weights [LMNY13, Yan17]. Given our strong negative result, we do not consider this commitment model any further.

Scheduling with commitment For scheduling with commitment upon admission, we give an (up to constant factors) optimal online algorithm with competitive ratio $\Theta\left(\frac{1}{\varepsilon}\right)$. For scheduling with δ-commitment, our result interpolates between the models commitment upon starting a job and commitment upon arrival. If $\delta \geq \frac{\varepsilon}{2}$, the competitive ratio is $\Theta\left(\frac{1}{\varepsilon}\right)$ which is best possible as we showed in Chapter 5. For $\delta \to \varepsilon$, the commitment requirement essentially implies commitment upon job arrival which has unbounded competitive ratio. Note that we give the first online algorithms for online throughput maximization with commitment on parallel identical machines with bounded competitive ratio for arbitrary slackness parameter ε.

Instances with arbitrary weights are hopeless without further restrictions. We show that there is no deterministic online algorithm with bounded competitive ratio for δ-commitment. Informally, our construction implies that there is no deterministic online algorithm with bounded competitive ratio in *any commitment model* in which a scheduler may have to commit to a job before it has completed. This is hard to formalize but may give guidance for the design of alternative commitment models. Our lower bound for δ-commitment is as follows: For $\delta, \varepsilon > 0$ with $\delta \leq \varepsilon < 1 + \delta$, no deterministic online algorithm has a bounded competitive ratio. In particular, this rules out bounded performance guarantees for $\varepsilon \in (0, 1)$. We remark that for sufficiently large slackness, i.e., $\varepsilon > 3$, Azar et al. [AKL+15] provide an online algorithm that has bounded competitive ratio. Our new lower bound answers affirmatively their open question whether high slackness is indeed required.

Finally, our impossibility result for weighted jobs and the positive result for instances without weights clearly separates the weighted from the unweighted setting.

Our techniques The challenge in online scheduling with commitment is that, once we committed to complete a job, the remaining slack of this job has to be spent very carefully. The key component of our algorithm is a job admission scheme which is implemented by different parameters. The high-level objectives are:

(i) Never start a job for the first time if its remaining slack is too small (parameter δ),

(ii) during the processing of a job, admit only significantly shorter jobs (parameter γ), and

(iii) for each admitted shorter job, block some time period (parameter β) during which no other jobs of similar size are accepted.

The first two goals are quite natural and have been used before (see Chapter 5 and [LMNY13]), while the third goal is crucial for our new tight result when scheduling with commitment. The intuition is the following: suppose we committed to complete a job with processing time 1 and have only a slack of $\mathcal{O}(\varepsilon)$ left before the deadline of this job. Suppose that c substantially smaller jobs of size $\frac{1}{c}$ arrive, where c is the competitive ratio we aim for. On the one hand, if we do not accept any of them, we cannot hope to achieve c-competitiveness. On the other hand, accepting too many of them fills up the slack and, thus, leaves no room for even smaller jobs. The idea is to keep the flexibility for future small jobs by only accepting an ε-fraction of jobs of similar size (within a factor two).

We distinguish two time periods with different regimes for accepting jobs. During the *scheduling interval* of job j, a more restrictive acceptance scheme ensures the completion of j whereas in the *blocking period* we guarantee the completion of previously accepted jobs. In contrast to the threshold algorithm in Chapter 5, where the processing time of the currently scheduled job provides a uniform acceptance threshold, this distinction enables us to ensure the completion of every admitted job without being too conservative in accepting jobs.

6.2 The Blocking Algorithm

In this section, we describe the *blocking algorithm* which handles scheduling with commitment. We assume that the slackness constant $\varepsilon > 0$ and, in the δ-commitment model, $\delta \in (0, \varepsilon)$ are given. If δ is not part of the input or if $\delta \leq \frac{\varepsilon}{2}$, then we set $\delta = \frac{\varepsilon}{2}$.

The algorithm never migrates jobs between machines, i.e., a job is only processed by the machine that initially started to process it. In this case, we say the job has been *admitted* to this machine. Moreover, our algorithm commits to completing a job upon admission. Hence, its remaining slack has to be spent very carefully on admitting other jobs to still be competitive. As our algorithm does not migrate jobs, it transfers the admission decision to the shortest admitted and not yet completed job on each machine. Thus, a job only admits significantly shorter jobs and prevents the admission of too many jobs of similar size. To this end, the algorithm maintains two types of intervals for each admitted job, a *scheduling interval* and a *blocking period*. A job can only be processed in its scheduling interval. Thus, it has to complete in this interval while admitting other jobs. Job j only admits jobs that are smaller by a factor of $\gamma = \frac{\delta}{16} < 1$. For an admitted job k, job j creates a blocking period of length at most βp_k, where $\beta = \frac{16}{\delta}$, which blocks the admission of similar-length jobs (cf. Figure 6.1). The scheduling intervals and blocking periods of jobs admitted by j will always be pairwise disjoint and completely contained in the scheduling interval of j.

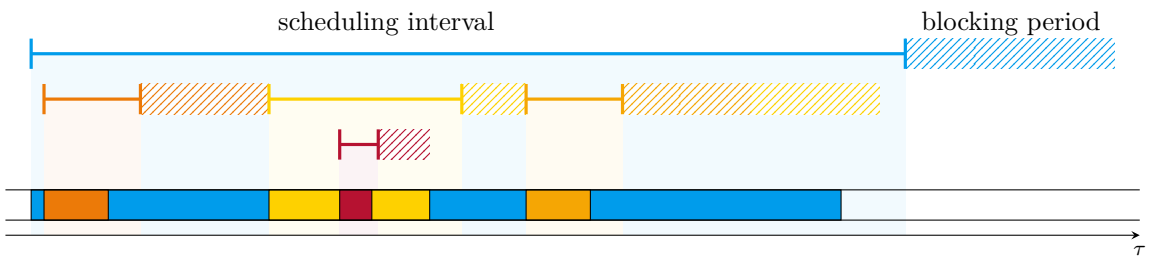

Figure 6.1: Scheduling interval, blocking period, and processing intervals

Scheduling jobs Similar to the threshold algorithm, the blocking algorithm follows the Shortest Processing Time (SPT) order for the set of uncompleted jobs assigned to a machine, which is independent of the admission scheme. SPT ensures that j has highest priority in the blocking periods of any job k admitted by j.

Admitting jobs The algorithm keeps track of *available* jobs at any time point τ. A job j with $r_j \leq \tau$ is called available if it has not yet been admitted to a machine by the algorithm and its deadline is not too close, i.e., $d_j - \tau \geq (1+\delta)p_j$.

Whenever a job j is available at a time τ and when there is a machine i such that time τ is not contained in the scheduling interval of any other job admitted to i, the shortest such job j is immediately admitted to machine i at time $a_j := \tau$, creating the scheduling interval $S(j) = [a_j, e_j)$, where $e_j = a_j + (1+\delta)p_j$ and an empty blocking period $B(j) = \emptyset$. In general, however, the blocking period is a finite union of time intervals associated with job j, and its size is the sum of lengths of the intervals, denoted by $|B(j)|$. Four types of events trigger a decision of the algorithm at time τ: the release of a job, the end of a blocking period, the end of a scheduling interval, and the admission of a job. In any of these four cases, the algorithm calls the class admission routine. This subroutine iterates over all machines i and checks if j, the shortest job on i whose scheduling interval contains τ, can admit the currently shortest available job $j^\star$.

To this end, any admitted job j checks if $p_{j^\star} < \gamma p_j$. Only such jobs qualify for admission by j. Upon admission by j, job $j^\star$ obtains two disjoint consecutive intervals, the *scheduling interval* $S(j^\star) = [a_{j^\star}, e_{j^\star})$ and the *blocking period* $B(j^\star)$ of size at most $\beta p_{j^\star}$. At the admission of job $j^\star$, the blocking period $B(j^\star)$ is planned to start at $e_{j^\star}$, the end of $j^\star$'s scheduling interval. During $B(j^\star)$, job j only admits jobs k with $p_k < \frac{1}{2}p_{j^\star}$.

Hence, when job j decides if it admits the currently shortest available job $j^\star$ at time τ, it makes sure that $j^\star$ is sufficiently small and that no job k of similar (or even smaller) processing time is blocking τ, i.e., it checks that $\tau \notin B(k)$ for all jobs k with $p_k \leq 2p_{j^\star}$ admitted to the same machine. In this case, we say that $j^\star$ is a *child* of j and that j is the *parent* of $j^\star$, denoted by $\pi(j^\star) = j$. If job $j^\star$ is admitted at time τ by job j, the algorithm sets $a_{j^\star} = \tau$ and $e_{j^\star} = a_{j^\star} + (1+\delta)p_{j^\star}$ and assigns the scheduling interval $S(j^\star) = [a_{j^\star}, e_{j^\star})$ to $j^\star$.

If $e_{j^\star} \leq e_j$, the routine sets $f_{j^\star} = \min\{e_j, e_{j^\star} + \beta p_{j^\star}\}$ which determines $B(j^\star) = [e_{j^\star}, f_{j^\star})$. As the scheduling and blocking periods of children k of j are supposed to be disjoint, we have to update the blocking periods. First consider the job k with $p_k > 2p_{j^\star}$ admitted to the same machine whose blocking period contains τ (if it exists), and let $[e'_k, f'_k)$ be the maximal interval of $B(k)$ containing τ. We set $f''_k = \min\{e_j, f'_k + (1+\delta+\beta)p_{j^\star}\}$ and replace the interval $[e'_k, f'_k)$ by $[e'_k, \tau) \cup [\tau + (1+\delta+\beta)p_{j^\star}, f''_k)$. For all other jobs k with $B(k) \cap [\tau, \infty) \neq \emptyset$ admitted to the same machine, we replace the remaining part of their blocking period $[e'_k, f'_k)$ by $[e'_k + (1+\delta+\beta)p_{j^\star}, f''_k)$ where $f''_k := \min\{e_j, f'_k + (1+\delta+\beta)p_{j^\star}\}$. In this update, we follow the convention that $[e, f) = \emptyset$ if $f \leq e$. Observe that the length of the blocking period might decrease due to such updates.

Note that $e_{j^\star} > e_j$ is also possible as j does not take the end of its own scheduling interval e_j into account when admitting jobs. Thus, the scheduling interval of $j^\star$ would end outside j's scheduling interval and inside j's blocking period. During $B(j)$, the parent $\pi(j)$ of j, did not allocate the interval $[e_j, e_{j^\star})$ for completing jobs admitted by j but for ensuring its own completion. Hence, the completion of both $j^\star$ and $\pi(j)$ is not necessarily guaranteed anymore. To prevent this, we modify all scheduling intervals $S(k)$ (including $S(j)$) that contain time τ of jobs admitted to the same machine as $j^\star$ and their blocking periods $B(k)$. For each job k admitted to the same machine with $\tau \in S(k)$ (including j) and $e_{j^\star} > e_k$ we set $e_k = e_{j^\star}$. We also update their blocking periods (in fact, single intervals) to reflect their new starting points. If the parent $\pi(k)$ of k does not exist, $B(k)$ remains empty; otherwise we set $B(k) := [e_k, f_k)$ where $f_k = \min\{e_{\pi(k)}, e_k + \beta p_k\}$. Note that, after this update, the blocking periods of any but the largest such job will be empty. Moreover, the just admitted job $j^\star$ does not get a blocking period in this special case.

During the analysis of the algorithm, we show that any admitted job j still completes before $a_j + (1+\delta)p_j$ and that $e_j \leq a_j + (1+2\delta)p_j$ holds in retrospect for all admitted jobs j. Thus, any job j that admits another job $j^\star$ tentatively assigns this job a scheduling interval of length $(1+\delta)p_{j^\star}$ but, for ensuring its own completion, it is prepared to lose $(1+2\delta)p_{j^\star}$ time units of its scheduling interval $S(j)$. We summarize the blocking algorithm in Algorithm 6.1.

Algorithm 6.1: Blocking algorithm

Scheduling routine: At all times τ and on all machines i, run the job with shortest processing time that has been admitted to i and has not yet completed .

Event: Upon release of a new job at time τ:
Call admission routine.

Event: Upon ending of a blocking period or scheduling interval at time τ:
Call admission routine.

Admission routine:
$j^\star \leftarrow$ a shortest available job at τ, i.e., $j^\star \in \arg\min\{p_j \mid j \in \mathcal{J}, r_j \leq \tau \text{ and } d_j - \tau \geq (1+\delta)p_j\}$
$i \leftarrow 1$
while $j^\star$ is not admitted and $i \leq m$ do

$K \leftarrow$ the set of jobs on machine i whose scheduling intervals contain τ
if $K = \emptyset$ do
    admit job $j^\star$ to machine i
        $a_{j^\star} \leftarrow \tau$ and $e_{j^\star} \leftarrow a_{j^\star} + (1+\delta)p_{j^\star}$
        $S(j^\star) \leftarrow [a_{j^\star}, e_{j^\star})$ and $B(j^\star) \leftarrow \emptyset$
    call admission routine
else
    $j \leftarrow \arg\min\{p_k \mid k \in K\}$
    if $j^\star < \gamma p_j$ and $\tau \notin B(j')$ for all j' admitted to i with $p_{j'} \leq 2p_{j^\star}$ do
        admit job $j^\star$ to machine i
            $a_{j^\star} \leftarrow \tau$ and $e_{j^\star} \leftarrow a_{j^\star} + (1+\delta)p_{j^\star}$
            if $e_{j^\star} \leq e_j$ do
                $f_{j^\star} \leftarrow \min\{e_j, e_{j^\star} + \beta p_{j^\star}\}$
                $S(j^\star) \leftarrow [a_{j^\star}, e_{j^\star})$ and $B(j^\star) \leftarrow [e_{j^\star}, f_{j^\star})$
            else
                $S(j^\star) \leftarrow [a_{j^\star}, e_{j^\star})$ and $B(j^\star) \leftarrow \emptyset$
                modify $S(k)$ and $B(k)$ for $k \in K$
        update $B(j')$ for j' admitted to machine i with $B(j') \cap [\tau, \infty) \neq \emptyset$
        call admission routine
    else
        $i \leftarrow i + 1$

Main Result and Road Map of the Analysis During the analysis, it is sufficient to concentrate on instances with small slack, as also noted in Chapter 5. For $\varepsilon > 1$, we run the blocking algorithm with $\varepsilon = 1$, which only tightens the commitment requirement, and obtain constant competitive ratios. Thus, we assume $0 < \varepsilon \leq 1$.

Theorem 6.1. *Consider throughput maximization on parallel identical machines with or without migration. There is an $\mathcal{O}\left(\frac{1}{\varepsilon - \delta'}\right)$-competitive online algorithm with commitment, where $\delta' = \frac{\varepsilon}{2}$ in the commitment-upon-admission model and $\delta' = \max\left\{\delta, \frac{\varepsilon}{2}\right\}$ in the δ-commitment model.*

We note that, in the δ-commitment model, committing to the completion of a job j at an earlier point in time clearly satisfies committing at a remaining slack of δp_j. Therefore, we may assume $\delta \in [\frac{\varepsilon}{2}, \varepsilon)$ and thus avoid dealing with δ'.

As in the previous chapter, we exploit that the blocking algorithm does not migrate any job. In other words, we compare again the throughput of our algorithm to the solution of an optimal non-migratory schedule. Then, we use the result by Kalyanasundaram and Pruhs [KP01, Theorem 1.1] on optimal migratory and non-migratory schedules to extend the analysis to the migratory setting; see Theorem 5.2.

The special structure of the blocking algorithm allows us again to split the proof of the result into two parts. The first part, Section 6.3, is to show that the blocking algorithm completes all admitted jobs on time. In the second part, Section 6.4, we show that the blocking algorithm

belongs to the class of online algorithms analyzed in Chapter 5 for bounding the throughput of an optimal, non-migratory solution. Then, our strong structural result (Theorem 5.9) enables us to prove that the blocking algorithm admits sufficiently many jobs to be competitive.

6.3 Completing All Admitted Jobs on Time

We show that the blocking algorithm finishes every admitted job on time in Theorem 6.3. As the blocking algorithm does not migrate jobs, it suffices to consider each machine individually in this section. The proof relies on the following observations: (i) The sizes of jobs admitted by job j that interrupt each others' blocking periods are geometrically decreasing, (ii) the scheduling intervals of jobs are completely contained in the scheduling intervals of their parents, and (iii) scheduling in SPT order guarantees that job j has highest priority in the blocking periods of its children. We start by proving the following technical lemma about the length of the final scheduling interval of an admitted job j, denoted by $|S(j)|$. In the proof, we use that $\pi(k) = j$ for two jobs j and k implies that $p_k < \gamma p_j$.

Lemma 6.2. *Let $0 < \delta < \varepsilon$ be fixed. If $\gamma > 0$ satisfies $(1 + 2\delta)\gamma \leq \delta$, then the length of the scheduling interval $S(j)$ of an admitted job j is upper bounded by $(1 + 2\delta)p_j$. Moreover, $S(j)$ contains the scheduling intervals and blocking periods of all descendants of j.*

Proof. By definition of the blocking algorithm, the end point e_j of the scheduling interval of job j is only modified when j or one of j's descendants admits another job. Let us consider such a case: If job j admits a job k whose scheduling interval does not fit into the scheduling interval of j, we set $e_j = e_k = a_k + (1 + \delta)p_k$ to accommodate the scheduling interval $S(k)$ within $S(j)$. The same modification is applied to any ancestor j' of j with $e_{j'} < e_k$. This implies that, after such a modification of the scheduling interval, neither j nor any affected ancestor j' of j are the smallest jobs in their scheduling intervals anymore. In particular, no job whose scheduling interval was modified in such a case at time τ is able to admit jobs after τ. Hence, any job j can only admit other jobs within the interval $[a_j, a_j + (1 + \delta)p_j)$. That is, $a_k \leq a_j + (1 + \delta)p_j$ for every job k with $\pi(k) = j$.

Thus, by induction, it is sufficient to show that $a_k + (1+2\delta)p_k \leq a_j + (1+2\delta)p_j$ for admitted jobs k and j with $\pi(k) = j$. Note that $\pi(k) = j$ implies $p_k < \gamma p_j$. Hence,

$$a_k + (1 + 2\delta)p_k \leq (a_j + (1 + \delta)p_j) + (1 + 2\delta)\gamma p_j \leq a_j + (1 + 2\delta)p_j,$$

where the last inequality follows from the assumption $(1 + 2\delta)\gamma \leq \delta$. Due to the construction of $B(k)$ upon admission of job k by job j, we also have $B(k) \subseteq S(j)$. □

Theorem 6.3. *Let $0 < \delta < \varepsilon$ be fixed. If $0 < \gamma < 1$ and $\beta \geq 1$ satisfy*

$$\frac{\beta/2}{\beta/2 + (1 + 2\delta)} \left(1 + \delta - 2(1 + 2\delta)\gamma\right) \geq 1, \tag{6.1}$$

then the blocking algorithm completes a job j admitted at $a_j \leq d_j - (1+\delta)p_j$ on time.

Our choice of parameters guarantees that Equation (6.1) is satisfied.

Proof. Let j be a job admitted by the blocking algorithm with $a_j \leq d_j - (1+\delta)p_j$. Showing that job j completes before time $d'_j := a_j + (1+\delta)p_j$ proves the theorem. Due to scheduling in SPT order, each job j has highest priority in its own scheduling interval if the time point does not belong to the scheduling interval of a descendant of j. Thus, it suffices to show that at most δp_j units of time in $[a_j, d'_j)$ belong to scheduling intervals $S(k)$ of descendants of j. By Lemma 6.2, the scheduling interval of any descendant k' of a child k of j is contained in $S(k)$. Hence, it is sufficient to only consider K, the set of children of j.

In order to bound the contribution of each child $k \in K$, we impose a *class structure* on the jobs in K depending on their size relative to job j. More precisely, we define $(\mathcal{C}_c(j))_{c\in\mathbb{N}_0}$ where $\mathcal{C}_c(j)$ contains all jobs $k \in K$ that satisfy $\frac{\gamma}{2^{c+1}}p_j \leq p_k < \frac{\gamma}{2^c}p_j$. As $k \in K$ implies $p_k < \gamma p_j$, each child of j belongs to exactly one class and $(\mathcal{C}_c(j))_{c\in\mathbb{N}_0}$ indeed partitions K.

Consider two jobs $k, k' \in K$ where, upon admission, k interrupts the blocking period of k'. By definition, we have $p_k < \frac{1}{2}p_{k'}$. Hence, the chosen class structure ensures that k belongs to a strictly *higher* class than k', i.e., there are $c, c' \in \mathbb{N}$ with $c > c'$ such that $k \in \mathcal{C}_c(j)$ and $k' \in \mathcal{C}_{c'}(j)$. In particular, the admission of a job $k \in \mathcal{C}_c(j)$ implies either that k is the first job of class $\mathcal{C}_c(j)$ that j admits or that the blocking period of the previous job in class $\mathcal{C}_c(j)$ has completed. Based on this distinction, we are able to bound the loss of scheduling time for j in $S(j)$ due to $S(k)$ of a child k.

Specifically, we partition K into two sets. The first set K_1 contains all children of j that where admitted as the first jobs in their class $\mathcal{C}_c(j)$. The set K_2 contains the remaining jobs.

We start with K_2. Consider a job $k \in \mathcal{C}_c(j)$ admitted by j. By Lemma 6.2, we know that $|S(k)| = (1+\mu\delta)p_k$, where $1 \leq \mu \leq 2$. Let $k' \in \mathcal{C}_c(j)$ be the previous job admitted by j in class $\mathcal{C}_c(j)$. Then, $B(k') \subseteq [e_{k'}, a_k)$. Since scheduling and blocking periods of children of j are disjoint, j has highest scheduling priority in $B(k')$. Hence, during $B(k') \cup S(k)$ job j is processed for at least $|B(k')|$ units of time. In other words, j is processed for at least a $\frac{|B(k')|}{|B(k')\cup S(k)|}$-fraction of $B(k') \cup S(k)$. We rewrite this ratio as

$$\frac{|B(k')|}{|B(k') \cup S(k)|} = \frac{\beta p_{k'}}{\beta p_{k'} + (1+\mu\delta)p_k} = \frac{\nu\beta}{\nu\beta + (1+\mu\delta)},$$

where $\nu := \frac{p_{k'}}{p_k} \in (\frac{1}{2}, 2]$. By differentiating with respect to ν and μ, we observe that the last term is increasing in ν and decreasing in μ. Thus, we lower bound this expression by

$$\frac{|B(k')|}{|B(k') \cup S(k)|} \geq \frac{\beta/2}{\beta/2 + (1+2\delta)}.$$

Therefore, j is processed for at least a $\frac{\beta/2}{\beta/2+(1+2\delta)}$-fraction in $\bigcup_{k\in K} B(k) \cup \bigcup_{k\in K_2} S(k)$.

We now consider the set K_1. The total processing volume of these jobs is bounded from above by $\sum_{c=0}^{\infty} \frac{\gamma}{2^c} p_j = 2\gamma p_j$. By Lemma 6.2, $|S(k)| \leq (1+2\delta)p_k$. Combining these two observations, we obtain $\left|\bigcup_{k\in K_1} S(k)\right| \leq 2(1+2\delta)\gamma p_j$. Combining the latter with the bound for K_2, we conclude that j is scheduled for at least

$$\left|[a_j, d'_j) \setminus \bigcup_{k\in K} S(k)\right| \geq \frac{\beta/2}{\beta/2 + (1+2\delta)}\Big((1+\delta) - 2(1+2\delta)\gamma\Big)p_j \geq p_j$$

units of time, where the last inequality follows from Equation (6.1). Therefore, j completes before $d'_j = a_j + (1+\delta)p_j \leq d_j$, which concludes the proof. □

6.4 Competitiveness: Admitting Sufficiently Many Jobs

After having proved that the blocking algorithm indeed completes all admitted jobs on time in the previous section, it remains to show that the blocking algorithm admits sufficiently many jobs to achieve the competitive ratio of $\mathcal{O}\left(\frac{1}{\varepsilon-\delta'}\right)$ where $\delta' = \frac{\varepsilon}{2}$ for commitment upon admission and $\delta' = \max\left\{\frac{\varepsilon}{2}, \delta\right\}$ for δ-commitment. To this end, we show that the blocking algorithm belongs to the class of online algorithms considered in Section 5.4.1. Then, Theorem 5.9 provides a bound on the throughput of an optimal non-migratory schedule. We restate the necessary properties of an online non-migratory algorithm $\mathcal{A}$ for convenience.

(P1) $\mathcal{A}$ only admits available jobs.

(P2) Retrospectively, for each time τ and each machine i, there is a threshold $u_{i,\tau} \in [0, \infty]$ such that any job j that was available and not admitted by $\mathcal{A}$ at time τ satisfies $p_j \geq u_{i,\tau}$ for every i.

(P3) The function $u^{(i)} : \mathbb{R} \to [0, \infty], \tau \mapsto u_{i,\tau}$ is piece-wise constant and right-continuous for every machine $i \in [m]$. Further, there are only countably many points of discontinuity.

The first property is clearly satisfied by the definition of the blocking algorithm. For the second and the third property, we observe that a new job $j^\star$ is only admitted to a machine i during the scheduling interval of another job j admitted to the same machine if $p_{j^\star} < \gamma p_j$. Further, the time point of admission must not be blocked by a similar- or smaller-size job k previously admitted during the scheduling interval of j. This leads to the bound $p_{j^\star} < \frac{1}{2}p_k$ for any job k whose blocking period contains the current time point. Combining these observations leads to a machine-dependent threshold $u_{i,\tau} \in [0, \infty]$ satisfying (P2) and (P3).

More precisely, fix a machine i and a time point τ. Using $j \to i$ to denote that j was admitted to machine i, we define $u_{i,\tau} := \min_{j:\, j\to i, \tau\in S(j)} \gamma p_j$ if there is no job k admitted to machine i with $\tau \in B(k)$. As usual, we have $\min \emptyset = \infty$. Otherwise, we set $u_{i,\tau} := \frac{1}{2}p_k$. We note that the function $u^{(i)}$ is piece-wise constant and right-continuous due to our choice

of right-open intervals for defining scheduling intervals and blocking periods. Moreover, the points of discontinuity of $u^{(i)}$ correspond to the admission of a new job, the end of a scheduling interval, and the start as well as the end of a blocking period of jobs admitted to machine i. Since we only consider instances with a finite number of jobs, there are at most finitely many points of discontinuity of $u^{(i)}$. Hence, we can indeed apply Theorem 5.9.

Then, the following theorem is the main result of this section.

Theorem 6.4. *An optimal non-migratory (offline) algorithm can complete at most a factor $\alpha+5$ more jobs on time than admitted by the blocking algorithm, where $\alpha := \frac{\varepsilon}{\varepsilon-\delta}\left(2\beta + \frac{1+2\delta}{\gamma}\right)$.*

Proof. We fix an instance and an optimal solution Opt. We use X to denote the set of jobs in Opt that the blocking algorithm did not admit. Without loss of generality, we can assume that all jobs in Opt complete on time. If J is the set of jobs admitted by the blocking algorithm, then $X \cup J$ is a superset of the jobs successfully finished in the optimal solution. Hence, showing $|X| \leq (\alpha+4)|J|$ suffices to prove Theorem 6.4.

We compare again the throughput of a highest loaded machine of the optimal solution to the throughput on a least loaded machine of the blocking algorithm. More precisely, let $\overline{X} \subseteq X$ be the jobs in Opt scheduled on a machine with *highest* throughput and let $\underline{J} \subseteq J$ be the jobs scheduled by the blocking algorithm on a machine with *lowest* throughput. With Theorem 5.9, we show $|\overline{X}| \leq (\alpha+4)|\underline{J}|$ to bound the cardinality of X in terms of $|J|$.

To this end, we retrospectively consider the interval structure created by the algorithm on the machine that schedules $\underline{J}$; let this without loss of generality be the first machine. Let $\underline{I}$ be the set of maximal intervals $I_t = [\tau_t, \tau_{t+1})$ such that $u_{1,\tau} = u_{1,\tau_t}$ for all $\tau \in I_t$. We define $\underline{u_t} = u_{1,\tau_t}$ for each interval I_t. As discussed above, the time points τ_t for $t \in [T]$ correspond to the admission, the end of a scheduling interval, and the start as well as the end of a blocking period of jobs admitted to machine 1. As the admission of a job adds at most three time points, we have that $|\underline{I}| \leq 3|\underline{J}| + 1$.

As the blocking algorithm satisfies Properties (P1) to (P3), we can apply Theorem 5.9 to obtain

$$|\overline{X}| \leq \sum_{t=1}^{T} \frac{\varepsilon}{\varepsilon-\delta} \frac{\tau_{t+1}-\tau_t}{\underline{u_t}} + |\underline{I}| \leq \sum_{t=1}^{T} \frac{\varepsilon}{\varepsilon-\delta} \frac{\tau_{t+1}-\tau_t}{\underline{u_t}} + (3|\underline{J}|+1).$$

It remains to bound the first part in terms of $|\underline{J}|$. If $\underline{u_t} < \infty$, let $j_t \in \underline{J}$ be the *smallest* job j with $\tau_t \in S(j) \cup B(j)$. Then, at most $\frac{\varepsilon}{\varepsilon-\delta}\frac{\tau_{t+1}-\tau_t}{\underline{u_t}}$ (potentially fractional) jobs will be charged to job j_t because of interval I_t. By definition of $\underline{u_t}$, we have $\underline{u_t} = \gamma p_{j_t}$ if $I_t \subseteq S(j_t)$, and if $I_t \subseteq B(j_t)$, we have $\underline{u_t} = \frac{1}{2}p_{j_t}$. The total length of intervals I_t for which $j = j_t$ holds sums up to at most $(1+2\delta)p_j$ for $I_t \subseteq S(j)$ and to at most $2\beta p_j$ for $I_t \subseteq B(j)$. Hence, in total, the charging scheme assigns at most $\frac{\varepsilon}{\varepsilon-\delta}(2\beta + \frac{1+2\delta}{\gamma}) = \alpha$ jobs in $\overline{X}$ to job $j \in \underline{J}$. Therefore,

$$|\overline{X}| \leq \Big(\alpha+3\Big)|\underline{J}| + 1.$$

If $\underline{J} = \emptyset$, the blocking algorithm admitted all jobs in the instance, and $|X| \leq |J|$ follows. Otherwise, $|\overline{X}| \leq \big(\alpha + 4\big)|\underline{J}|$, and we obtain

$$|\textsc{Opt}| \leq |X \cup J| \leq m|\overline{X}| + |J| \leq m(\alpha+4)|\underline{J}| + |J| \leq (\alpha+5)|J|,$$

which concludes the proof. □

Finalizing the proof of Theorem 6.1

Proof of Theorem 6.1. In Theorem 6.3 we show that the blocking algorithm completes all admitted jobs J on time. This implies that the blocking algorithm is feasible for the model commitment upon admission. As no job $j \in J$ is admitted later than $d_j - (1+\delta)p_j$, the blocking algorithm also solves scheduling with δ-commitment. Theorem 1.1 in [KP01] (Theorem 5.2) gives a bound on the throughput of an optimal migratory schedule in terms of the throughput of an optimal non-migratory solution. In Theorem 6.4, we bound the throughput $|\textsc{Opt}|$ of an optimal non-migratory solution by $|J|$, the throughput of the blocking algorithm. Combining these theorems shows that the blocking algorithm achieves a competitive ratio of

$$c = 6(\alpha+5) = 6\left(\frac{\varepsilon}{\varepsilon-\delta}\Big(2\beta + \frac{1+2\delta}{\gamma}\Big) + 5\right).$$

Our choice of parameters $\beta = \frac{16}{\delta}$ and $\gamma = \frac{\delta}{16}$ implies $c \in \mathcal{O}\Big(\frac{\varepsilon}{(\varepsilon-\delta)\delta}\Big)$. For commitment upon arrival or for δ-commitment in the case where $\delta \leq \frac{\varepsilon}{2}$, we run the algorithm with $\delta' = \frac{\varepsilon}{2}$. Hence, $c \in \mathcal{O}\Big(\frac{1}{\varepsilon-\delta'}\Big) = \mathcal{O}\Big(\frac{1}{\varepsilon}\Big)$. If $\delta > \frac{\varepsilon}{2}$, then we set $\delta' = \delta$ in our algorithm. Thus, $\frac{\varepsilon}{\delta'} \in \mathcal{O}(1)$ and, again, $c \in \mathcal{O}\Big(\frac{1}{\varepsilon-\delta'}\Big)$. □

6.5 Lower Bounds on the Competitive Ratio

We emphasize that the blocking algorithm matches the lower bound presented in the previous chapter for online throughput maximization when scheduling without commitment. In this section, we give an impossibility result even for randomized algorithms for scheduling with commitment upon arrival. Since the δ-commitment requirement essentially tightens to commitment upon arrival if δ converges to ε, the divergence of the competitive ratio of the blocking algorithm for $\delta \to \varepsilon$ is justified.

Further, we develop several lower bounds for scheduling with commitment in the presence of weights.

Commitment Upon Arrival

We substantially strengthen earlier results for weighted jobs [LMNY13, Yan17] and show that the model is hopeless even in the unweighted setting and even for randomized algorithms.

Theorem 6.5. *No randomized online algorithm has a bounded competitive ratio for commitment upon arrival.*

In the proof of the theorem, we use the following algebraic fact.

Lemma 6.6. *If some positive numbers $q_1, \dots, q_k, c \in \mathbb{R}_+$ satisfy the properties*
(i) $\sum_{\ell=1}^{k} q_\ell \leq 1$ *and*
(ii) $\sum_{\ell=1}^{j} q_\ell \cdot 2^{\ell-1} \geq \frac{2^{j-1}}{c}$ *for all* $j = 1, \dots, k$,
then $c \geq \frac{k+1}{2}$.

Proof. We take a weighted sum over all inequalities in (ii), where the weight of the inequality corresponding to $j < k$ is 2^{k-j-1} and the weight of the inequality corresponding to $j = k$ is 1. The result is

$$\sum_{\ell=1}^{k} q_\ell \cdot 2^{k-1} \geq \frac{(k+1) \cdot 2^{k-2}}{c} \Leftrightarrow \sum_{\ell=1}^{k} q_\ell \geq \frac{(k+1)}{2c}.$$

If $c < \frac{k+1}{2}$, this contradicts (i). □

We proceed to the proof of the theorem.

Proof of Theorem 6.5. Consider any $\varepsilon > 0$ and an arbitrary $\gamma \in (0, 1)$. Toward a contradiction, suppose that there is a (possibly randomized) c-competitive algorithm, where c may depend on ε.

Let $k \in \mathbb{N}$ with $k \geq 2c$. The instance consists of one machine and at most k *waves* of jobs, but the instance may end after any wave.

Wave ℓ has 2^ℓ jobs. Each job from the ℓth wave has release date $\frac{\ell}{k} \cdot \gamma$, deadline 1, and processing time $\frac{1}{2^\ell} \cdot \frac{1-\gamma}{1+\varepsilon}$. Choosing $p_j \leq \frac{1-\gamma}{1+\varepsilon}$ for all jobs j ensures that $d_j - r_j \geq (1+\varepsilon)p_j$. Further, note that the total volume of jobs in wave ℓ adds up to no more than $1 - \gamma$.

Define q_ℓ to be the expected total processing time of jobs that the algorithm accepts from wave ℓ. We observe:

(i) Since all accepted jobs have to be scheduled within the interval $[0, 1]$, we must have $\sum_{\ell=1}^{k} q_\ell \leq 1$.

(ii) For each ℓ, possibly no further jobs are released after wave ℓ. Since, in this case, the optimum schedules all jobs from wave ℓ and the jobs' processing times decrease by a factor of 2 from wave to wave, it must hold that $\sum_{\ell=1}^{j} q_\ell \cdot 2^{\ell-1} \geq \frac{2^{j-1}}{c}$ for all $j \in [k]$.

This establishes the conditions of Lemma 6.6 for $q_1, \dots, q_k$, which implies $c \geq \frac{k+1}{2} > c$. This gives a contradiction. □

Commitment on Job Admission and δ-commitment.

Since scheduling with commitment is more restrictive than scheduling without commitment, the lower bound $\Omega(\frac{1}{\varepsilon})$ from Theorem 5.12 holds for throughput maximization with commitment upon job admission and δ-commitment.

In the remainder of this section, we consider weighted throughput maximization where jobs may have arbitrary weights or where the weights are equal to their processing times.

Commitment upon admission For scheduling with arbitrary weights, Azar et al. [AKL+15] rule out any bounded competitive ratio for deterministic algorithms. Thus, our bounded competitive ratio for the unweighted setting (Theorem 6.1) gives a clear separation between the weighted and the unweighted setting.

Scheduling with δ-commitment We give a lower bound depending on parameters ε and δ.

Theorem 6.7. *Consider scheduling weighted jobs in the δ-commitment model. For $\delta, \varepsilon > 0$ with $\delta \leq \varepsilon < 1 + \delta$, no deterministic online algorithm has a bounded competitive ratio.*

Proof. We reuse the idea of [AKL+15] to release the next job upon admission of the previous one while heavily increasing the weights of subsequent jobs. However, the scheduling models differ in the fact that the δ-commitment model allows for processing before commitment which is not allowed in the commitment-upon-admission model.

Toward a contradiction, suppose that there is a c-competitive algorithm. We consider the following instance with one machine and n jobs with the same deadline d, where $d = 1 + \varepsilon$. Job $j \in [n]$ has weight $(c+1)^j$ which implies that any c-competitive algorithm has to admit job j at some point even if all jobs $1, \dots, j-1$ are admitted. In the δ-commitment model, the commitment to job j cannot happen later than $d - (1+\delta)p_j$, which is shortly before the release date of job $j+1$.

More precisely, the first job is released at $r_1 = 0$ with processing time $p_1 = 1$. If jobs $1, \dots, j$ have been released, then job $j+1$ is released at $r_{j+1} = d - (1+\delta)p_j + \varphi p_j$, for $\varphi \in (0, \delta)$, and has processing time

$$p_{j+1} = \frac{d - r_{j+1}}{1+\varepsilon} = \frac{d - (d - (1+\delta)p_j + \varphi p_j)}{1+\varepsilon} = \frac{1+\delta-\varphi}{1+\varepsilon} p_j = \left(\frac{1+\delta-\varphi}{1+\varepsilon}\right)^j.$$

An instance with n such jobs has a total processing volume of

$$\sum_{j=1}^{n} p_j = \sum_{j=0}^{n-1} \left(\frac{1+\delta-\varphi}{1+\varepsilon}\right)^j = \frac{1 - \left(\frac{1+\delta-\varphi}{1+\varepsilon}\right)^n}{1 - \frac{1+\delta-\varphi}{1+\varepsilon}}.$$

Any c-competitive algorithm has to complete the n jobs before $d = 1 + \varepsilon$. This also holds for $n \to \infty$ and $\varphi \to 0$, and thus $\frac{1+\varepsilon}{\varepsilon-\delta} \leq 1 + \varepsilon$ is implied. This is equivalent to $\varepsilon \geq 1 + \delta$. In other words, if $\varepsilon < 1 + \delta$, there is no deterministic c-competitive online algorithm. □

In particular, there is no bounded competitive ratio possible for $\varepsilon \in (0, 1)$. A restriction of ε appears to be necessary since Azar et al. [AKL+15] provide such a bound when the slackness is sufficiently large, i.e, $\varepsilon > 3$. In fact, our bound answers affirmatively the open question

in [AKL+15] whether or not high slackness is indeed required. Again, this strong impossibility result gives a clear separation between the weighted and the unweighted problem as we show in the unweighted setting a bounded competitive ratio for any $\varepsilon > 0$ (Theorem 6.1).

Proportional weights For scheduling with commitment, it is known that simple greedy algorithms achieve the best possible competitive ratio $\Theta\left(\frac{1}{\varepsilon}\right)$ [DP00, GNYZ02]. In this section, we show a weaker lower bound for randomized algorithms.

Theorem 6.8. *Consider proportional weights ($w_j = p_j$). For commitment on job admission and the δ-commitment model, the competitive ratio of any randomized algorithm is $\Omega\left(\log \frac{1}{\varepsilon}\right)$.*

Proof. Let $k = \left\lfloor \log\left(\frac{1}{8\varepsilon}\right)\right\rfloor$, and consider a c-competitive algorithm. The instance consists of one machine and at most k jobs, where job $j \in [k]$ arrives at $2\varepsilon \sum_{\ell=1}^{j-1} 2^{\ell-1}$ and has processing time 2^{j-1} and slack $\varepsilon 2^{j-1}$. The release date of job j is

$$2\varepsilon \sum_{\ell=1}^{j-1} 2^{\ell-1} < 2\varepsilon \cdot 2^{\log(1/(8\varepsilon))} \leq \frac{1}{4},$$

at which time any job $j' < j$ that the algorithm has committed to has at least $p_1 - \frac{1}{4} = \frac{3}{4}$ units of processing time left. However, the slack of j is at most

$$\varepsilon \cdot 2^{j-1} \leq \varepsilon \cdot 2^{\lfloor \log(1/(8\varepsilon))\rfloor - 1} \leq \frac{1}{16}.$$

This implies that no algorithm should commit to two jobs at the same time. If q_ℓ is the probability that the algorithm commits to job ℓ, then $\sum_{\ell=1}^{k} q_\ell \leq 1$.

Further, if the algorithm commits to $j < k$, then this has to happen at the latest at time

$$r_j + \varepsilon 2^{j-1} = 2\varepsilon \sum_{\ell=1}^{j-1} 2^{\ell-1} + \varepsilon 2^{j-1} < 2\varepsilon \sum_{\ell=1}^{j} 2^{\ell-1} = r_{j+1}.$$

That is, unknowing whether $j+1$ will be released or not, the algorithm has to be competitive with the optimum that only schedules job j. As such an optimum achieves a value of $p_j = 2^{j-1}$, any c-competitive algorithm has to satisfy $\sum_{\ell=1}^{j} q_\ell \cdot 2^{\ell-1} \geq \frac{2^{j-1}}{c}$.

Therefore, we are able to apply Lemma 6.6 to $q_1, \ldots, q_k$, showing $c \geq \frac{k+1}{2} = \Omega\left(\log \frac{1}{\varepsilon}\right)$. □

6.6 Concluding Remarks

We answer the major open questions regarding online throughput maximization with commitment requirements and give an optimal online algorithm on identical parallel machines for the problem $P \,|\, \text{online } r_j, \text{pmtn} \,|\, \sum(1-U_j)$ when scheduling with commitment upon admission or with δ-commitment. Surprisingly, the asymptotic performance of an online scheduler does

not change significantly under these moderate, yet valuable commitment requirements. For the most restrictive model, commitment upon arrival, we rule out any online algorithm with bounded competitive ratio.

As observed in the previous chapter, our lower bounds on the competitive ratio are based on single-machine instances. Hence, it remains open whether the problem where m is not part of the input admits an online algorithm with a better competitive ratio as is the case for $Pm \,|\, \text{online } r_j, \text{pmtn} \,|\, \sum p_j(1 - U_j)$ [SS16].

7

Dynamic Multiple Knapsacks

In the Multiple Knapsack problem, we are given multiple knapsacks with different capacities and items with values and sizes. The task is to find a subset of items of maximum total value that can be packed into the knapsacks without exceeding the capacities. We investigate this problem and special cases thereof in the context of *dynamic algorithms* and design data structures that efficiently maintain near-optimal knapsack solutions for dynamically changing input. More precisely, we handle the arrival and departure of individual items or knapsacks during the execution of the algorithm with worst-case update time poly-logarithmic in the number of items. As an optimal and any approximate solution may change drastically with changing input, we only maintain implicit solutions and support certain queries in poly-logarithmic time, such as asking for the packing of an item or the solution value.

While dynamic algorithms are well-studied in the context of graph problems, there is hardly any work on packing problems and generally much less on non-graph problems. Given the theoretical interest in knapsack problems and their practical relevance, it is somewhat surprising that Knapsack has not been addressed before in the context of dynamic algorithms. Our work bridges this gap.

Bibliographic Remark: This chapter is based on joint work with M. Böhm, N. Megow, L. Nölke, J. Schlöter, B. Simon, and A. Wiese [BEM+20]. Therefore, some parts correspond to or are identical with [BEM+20], which is submitted for publication at SODA 2021. The proofs of Sections 7.6 and 7.7 will (also) appear in the PhD thesis by L. Nölke.

Table of Contents

7.1 Introduction

Knapsack problems are among the most fundamental optimization problems, studied since the early days of optimization theory. In the most basic variant, the Knapsack problem, there are given a knapsack with capacity $S \in \mathbb{N}$ and a set $\mathcal{J}$ of n items, where $\mathcal{J} = [n]$, and each item j has a size $s_j \in \mathbb{N}$ and a value $v_j \in \mathbb{N}$. The goal is to find a subset of items, $P \subseteq [n]$, with maximal total value $v(P) = \sum_{j \in P} v_j$, and with total size $s(P) = \sum_{j \in P} s_j$, that does not exceed the knapsack capacity S. In the more general Multiple Knapsack problem, we are given m knapsacks with capacities S_i for $i \in [m]$. Here, the task is to select m disjoint subsets $P_1, \ldots, P_m \subseteq \mathcal{J}$ such that subset P_i satisfies the capacity constraint $s(P_i) \leq S_i$ and the total value of all subsets $\sum_{i=1}^{m} v(P_i)$ is maximized.

The Knapsack problem is $\mathcal{NP}$-complete in its decision variant — in fact, it is one of the 21 problems on Karp's list of $\mathcal{NP}$-complete problems [Kar72] — and it admits pseudo-polynomial time algorithms. The first published pseudopolynomial-time algorithm for Knapsack from the 1950s has running time $\mathcal{O}(n \cdot S)$ [Bel57]. The decision variant of Multiple Knapsack is strongly $\mathcal{NP}$-complete, even for identical knapsack capacities, since it is a special case of Bin Packing [GJ79, KPP04]. Hence, it does not admit pseudopolynomial-time algorithms, unless $\mathcal{P} = \mathcal{NP}$.

As a consequence of these hardness results, each of the knapsack variants has been studied extensively over the years through the lens of approximation algorithms. Of particular interest are *approximation schemes*, families of polynomial-time algorithms that compute for any constant $\varepsilon > 0$ a $(1+\varepsilon)$-approximate solution, i.e., a feasible solution with value within a factor of $(1+\varepsilon)$ of the optimal solution value (see also Chapter 2). The first approximation scheme for the Knapsack problem is due to Ibarra and Kim [IK75] and has running time polynomial in n and $\frac{1}{\varepsilon}$. This seminal paper initiated a long sequence of follow-up work, with the latest improvements appearing only recently [Cha18, Jin19].

Multiple Knapsack is substantially harder and does not admit $(1+\varepsilon)$-approximate algorithms with running time polynomial in $\frac{1}{\varepsilon}$, unless $\mathcal{P} = \mathcal{NP}$, even with two identical knapsacks [CK05]. However, some approximation schemes with exponential dependency on $\frac{1}{\varepsilon}$ are known [Kel99, CK05] as well as improved variants, where the dependency on $f\left(\frac{1}{\varepsilon}\right)$ for some function f is only multiplicative or additive [Jan09, Jan12]. The currently fastest known approximation scheme has a running time of $2^{\mathcal{O}(\log^4(1/\varepsilon)/\varepsilon)} + \text{poly}(n)$ [Jan12]. All these algorithms are *static* in the sense that the algorithm has access to the entire instance, and the instance is not subject to changes.

The importance of knapsack problems in theory and practice is reflected by the two dedicated books [MT90, KPP04]. Given the relevance of knapsack applications in practice and the ubiquitous dynamics of real-world instances, it is natural to ask for *dynamic algorithms* that adapt to small changes in the packing instance while spending only little time to recompute the solution. More precisely, during the execution of the algorithm, items and knapsacks arrive and depart, and the algorithm has to maintain an approximate knapsack solution with a small update time, preferably poly-logarithmic in the current number of items. A dynamic algorithm for knapsack problems can be seen as a data structure that supports *update operations* to insert or remove an item or a knapsack as well as relevant *query operations* to output the current solution. We use *update time* to refer to the running time that is needed to update the underlying data structure and to compute the new solution. We are the first to analyze knapsack problems in the context of dynamic algorithms.

Generally, dynamic algorithms constitute a vibrant research field in the context of graph problems. We refer to the surveys [DEGI10, Hen18, BP11] for an overview on dynamic graph algorithms. For packing and, generally, for non-graph-related problems, dynamic algorithms with small update time are much less studied. A notable exception is a result for Bin Packing that maintains a $\frac{5}{4}$-approximate solution with $\mathcal{O}(\log n)$ update time [IL98]. This lack of efficient dynamic algorithms is in stark contrast to the aforementioned intensive research on computationally efficient algorithms for knapsack problems. Our work bridges this gap initiating the design of algorithms that efficiently maintain near-optimal solutions.

Our Contribution

In this chapter, we present dynamic algorithms for maintaining approximate knapsack solutions for two problems of increasing complexity: Multiple Knapsack with identical knapsack sizes and Multiple Knapsack without further restrictions. Our algorithms are *fully dynamic* which means that in an update operation they can handle both, the arrival or departure of an item and the arrival or departure of a knapsack. Further, we consider the *implicit solution* or *query* model, in which an algorithm is not required to store the solution explicitly in memory such that the solution can be read in linear time at any given point of the execution. Instead, the algorithm may maintain the solution implicitly with the guarantee that a query about the

packing can be answered in poly-logarithmic time. Since KNAPSACK is already $\mathcal{NP}$-hard even with full knowledge of the instance, we aim at maintaining $(1+\varepsilon)$-approximate solutions.

We give *worst-case* guarantees for update and query times that are poly-logarithmic in n, the number of items currently in the input, and bounded by a function of $\varepsilon > 0$, the desired approximation accuracy. For some special cases, we can even ensure a polynomial dependency on $\frac{1}{\varepsilon}$. In others, we justify the exponential dependency with $\mathcal{NP}$-hardness results. Denote by $v_{\max}$ the currently largest item value and by $S_{\max}$ the currently largest knapsack capacity.

- For MULTIPLE KNAPSACK, we design a dynamic algorithm that maintains a $(1+\varepsilon)$-approximate solution with update time $2^{f(1/\varepsilon)}\left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}(\log m \log S_{max} \log v_{\max})^{\mathcal{O}(1)}$, where $f(1/\varepsilon)$ is quasi-linear, and query time $\mathcal{O}\left(\frac{\log n}{\varepsilon^2}+\log m\right)$ for single items (Section 7.7).
- The exponential dependency on $\frac{1}{\varepsilon}$ in the update time for MULTIPLE KNAPSACK is indeed necessary, even for two identical knapsacks. We show that there is no $(1+\varepsilon)$-approximate dynamic algorithm with update time $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1)}$, unless $\mathcal{P}=\mathcal{NP}$ (Section 7.2).
- For MULTIPLE KNAPSACK with m *identical knapsacks*, we maintain a $(1+\varepsilon)$-approximate solution with update time $\left(\frac{1}{\varepsilon}\log n \log S_{max} \log v_{\max}\right)^{\mathcal{O}(1)}$ and query time $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1)}$ if $m \geq \frac{16}{\varepsilon^7}\log^2 n$ (Section 7.4). For small m, we refer to Section 7.6 for a high-level overview and to [BEM+20] for the details.

In each update step, we compute only implicit solutions and provide queries for the solution value, the knapsack of a queried item, or the complete solution. These queries are consistent between two update steps and run efficiently, i.e., polynomially in $\log n$ and $\log v_{\max}$ and with a dependency on ε and the output size. We remark that it is not possible to maintain a solution with a non-trivial approximation guarantee explicitly with only poly-logarithmic update time (even amortized) since it might be necessary to change $\Omega(n)$ items per iteration, e.g., if a very large and very profitable item is inserted and removed in each iteration. Therefore, instead of packing an item implicitly, we transform items into types and for those, we only store slots that are then filled with items of the correct type upon query.

Methodology

Dynamic linear grouping We develop this technique to cluster a (sub)set of items into so-called *item types* of roughly the same size and value in time $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1)}$. Traditionally, linear grouping is applied for solving bin packing problems, where any feasible solution has to pack all items [dlVL81]. This property is crucial since the cardinality of the groups depends on the number of packed items. In knapsack problems, however, a feasible solution may consist of only a subset of items. We handle this uncertainty by simultaneously executing classical linear grouping for $\mathcal{O}(\log_{1+\varepsilon} n)$ many guesses of the cardinality of an optimal solution, and thus we simulate the possible choices which subset to select; see Section 7.3.

Identical knapsacks As a special case, we consider Multiple Knapsack with identical capacities in the dynamic setting. We call an item type *small* or *big* if its size is at most or at least an ε-fraction of the knapsacks' capacity, respectively. As the number of big items per knapsack is bounded, we use a configuration integer linear program (ILP) to explicitly assign these items via configurations to knapsacks. Conversely, the ILP assumes that small items can be packed fractionally and thus assigns those only via a placeholder. Even after applying dynamic linear grouping, the number of variables is still prohibitively large. Hence, we would like to apply the Ellipsoid Method with an approximate separation oracle to the dual problem similar to its application in [KK82, PST95, Rot12]. However, we cannot use any of their approaches directly due to two additional variables in the dual problem. Instead, we add an objective function constraint to the dual problem and carefully exploit the connection between feasible and infeasible dual solutions to obtain a basic feasible solution for the primal problem. This enables us to approximately solve the LP relaxation and round the so found solution in time $\left(\frac{1}{\varepsilon}\log n \log S_{max} \log v_{\max}\right)^{\mathcal{O}(1)}$ if m is sufficiently large; see Section 7.4.

Multiple Knapsack We design a dynamic algorithm for Multiple Knapsack with update time $\left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}(\log m \log S_{max} \log v_{\max})^{\mathcal{O}(1)}$. We accomplish this goal by partitioning the given knapsacks based on their capacity, creating two subproblems of Multiple Knapsack. This separation allows us to design algorithms that exploit the structural properties specific to each subproblem. One subproblem consists of relatively few (though non-constantly many) knapsacks, but they are the largest of the instance. While the small number of these *special* knapsacks offers more algorithmic freedom, this freedom is necessary since great care has to be taken when computing a solution. After all, there may be items of high value that only fit into special knapsacks. The second subproblem contains almost all remaining smaller knapsacks. The sheer number of these *ordinary* knapsacks results in a reversed problem, with the algorithmic handling of the numerous knapsacks being a major hurdle. On the upside, mistakes are forgiven more easily, allowing us to even discard a small fraction of knapsacks entirely. Additionally, we create a third partition of knapsacks that lies in-between the two subproblems (with respect to knapsack capacity). It consists of knapsacks that contribute negligible value to an optimal solution. This property induces the precise partitioning and allows us to consider the knapsacks as empty *extra* knapsacks, which we use to place leftover items not packed in the subproblems.

The major challenge with this divide-and-conquer approach is to decide which item is assigned to which of the two subproblems. Clearly, for some — *special* — items this question is answered by their size as they only fit into special knapsacks, unlike the remaining — *ordinary* — items. In fact, for them the allocation is so problematic that we resort to downright putting a number of high-value ordinary items into extra knapsacks. To handle the remainder, we guess the total size of ordinary items that are put into special knapsacks by an optimal solution. We then add a virtual knapsack — with capacity equal to this guess — to the ordinary

subproblem and solve it with the not yet packed ordinary items as input. The input for the special subproblem then consists of all special items together with bundles of the ordinary items packed in the virtual knapsack. In Section 7.5, we explain in detail how the ordinary subproblem is solved while Section 7.6 gives an overview of the special subproblem.

Related Work

Ever since the first approximation scheme for Knapsack due to Ibarra and Kim [IK75], running times have been improved steadily over the last decades [GL79, Law79, GL80, KP04, Rhe15, Cha18, Jin19] with $\mathcal{O}\left(n \log \frac{1}{\varepsilon} + \left(\frac{1}{\varepsilon}\right)^{9/4}\right)$ by Jin [Jin19] currently being the fastest. Recent work on conditional lower bounds implies that Knapsack does not admit an FPTAS with running time $\mathcal{O}\left(\left(n + \frac{1}{\varepsilon}\right)^{2-\delta}\right)$, for any $\delta > 0$, unless $(\min, +)$-convolution has a subquadratic algorithm [CMWW19, MWW19].

A PTAS for Multiple Knapsack was first discovered by Chekuri and Khanna [CK05] and an EPTAS due to Jansen [Jan09] is also known. The running time of this EPTAS is $2^{\mathcal{O}(\log(1/\varepsilon)/\varepsilon^5)} \cdot \text{poly}(n)$. Jansen [Jan12] later presented a second EPTAS with an improved running time of $2^{\mathcal{O}(\log^4(1/\varepsilon)/\varepsilon)} + \text{poly}(n)$. These algorithms are all static and do not explicitly support efficient update operations except when being run from scratch after each update. Hence, directly applying such an approximation scheme after each update is prohibitive since a single item arrival can change a packing solution completely, requiring a full recomputation with running time polynomial in the input size.

At the heart of the two EPTASs [Jan09, Jan12] lies a configuration ILP for rounded items and/or knapsacks of exponential size. Even though near-optimal solutions to the LP relaxation can be found and rounded in time $\mathcal{O}(\text{poly}(n))$, this is beyond the scope of the poly-logarithmic update time we are interested in. Additionally, the configuration ILPs still contain $\Omega(n)$ many constraints and variables which is yet another obstacle when aiming for dynamically maintaining approximate solutions with poly-logarithmic running time. Hence, to improve the running time according to our goal of poly-logarithmic update time, a more careful approach for rounding items has to be developed before similar configuration ILPs can be applied.

7.2 Data Structures and Preliminaries

From the perspective of a data structure that implicitly maintains near-optimal solutions for Multiple Knapsack, our algorithms support several different update and query operations. These allow for the input to Multiple Knapsack to change, which causes the computation of a new solution, or for (parts of) that solution to be output, respectively. The supported *update operations* are as follows.

- **Insert Item:** inserts an item into the input

- **Remove Item j:** removes item j from the input
- **Insert Knapsack:** inserts a knapsack into the input
- **Remove Knapsack i:** removes knapsack i from the input

These update operations compute a new solution which can be output, entirely or in parts, using the following *query operations*.

- **Query Item j:** returns whether item j is packed in the current solution and if so, additionally returns the knapsack containing it
- **Query Solution Value:** returns the value of the current solution
- **Query Entire Solution:** returns all items in the current solution, together with the information in which knapsack each such item is packed

Since the solution is allowed to change only *after* an update, these queries are consistent in-between two update operations. Nevertheless, the answers to queries are not independent of each other but depend on the precise order of the queries. This is mostly due to our approach of reserving slots for items of a particular type and filling these slots with items explicitly only upon query.

To provide the above functionality, we require the use of additional data structures and make a few basic assumptions which we now discuss. First, while the model imposes no time bounds on the computation of an initial solution, we can compute such an initial solution by inserting one item/knapsack at a time and computing the implicit solution after all the insertions in time nearly linear in n and m and with additional dependencies on ε and $v_{\max}$ as in the respective algorithms. For simplicity, we assume that elementary operations such as addition, multiplication, and comparison of two values can be handled in constant time. Clearly, this is not true as the parameters involved can be as large as $v_{\max}$ and $S_{\max}$. However, as we will show, the number of elementary operations is bounded, and thus their results do not grow arbitrarily large but are in fact bounded by a polynomial in $\log n$, $\log m$, $S_{\max}$, and $v_{\max}$ and some function of $\frac{1}{\varepsilon}$. Thus, we do not explicitly state the size of the involved numbers. Lastly, we make some standard assumptions on ε. By appropriately decreasing ε, we assume without loss of generality that $\frac{1}{\varepsilon} \in \mathbb{N}$ and $\varepsilon \leq 1$. Further, we present our results in the form of $(1 + \mathcal{O}(\varepsilon))$-approximate algorithms to simplify the exposition. For achieving the required approximation guarantee of $1+\varepsilon$, we can appropriately choose some $\varepsilon' \in \Theta(\varepsilon)$ for running the algorithms without changing the asymptotic dependency of the running time on ε.

Rounding Values A crucial ingredient to our algorithms is the partitioning of items into only few *value classes* V_ℓ consisting of items j for which $(1+\varepsilon)^\ell \leq v_j < (1+\varepsilon)^{\ell+1}$, for $\ell \in \mathbb{N}_0$. Upon arrival of an item, we calculate its value class V_{ℓ_j} and store j together with v_j, s_j, and ℓ_j in

the appropriate data structures of the respective algorithm. We assume all items in V_ℓ to have value $(1+\varepsilon)^\ell$. Since this technique is rather standard, we do not provide a formal proof of the next lemma but only give its statement.

Lemma 7.1. *(i) There are at most* $\mathcal{O}\left(\frac{\log v_{\max}}{\varepsilon}\right)$ *many value classes.*

(ii) For optimal solutions Opt *and* Opt′ *to the original and rounded instance respectively, we have* $(1+\varepsilon)v(\text{Opt}') \geq v(\text{Opt})$.

Data Structures The targeted running times do not allow for completely reading the instance in every round but rather ask for carefully maintained data structures that allow us to quickly compute and store implicit solutions. For access to the input, we maintain an array that for each item stores the item's size, value, and value class, and similarly for knapsacks. However, our dynamic algorithms mostly rely on maintaining sorted lists of up to n items or m knapsacks, respectively. For all orderings, break ties according to indices. For sorting the items, we will mostly use their size or their *density*, the ratio between the value v_j and the size s_j of an item $j \in \mathcal{J}$.

Since our goal is to design algorithms with poly-logarithmic update times, it is crucial that the data structures enable accordingly efficient insertion, deletion, and access times. Bayer and McCreight [BM72] developed such a data structure in 1972, the so-called B-trees that were later refined by Bayer [Bay72] to *symmetric binary B-trees*. These trees store elements sorted according to some key value in their nodes. In contrast to this early work, in each node k, we additionally store information about the total size, the total value, the total number, or the total capacity of the elements in the subtree rooted at k.

As observed by Olivié [Oli82] and by Tarjan [Tar83], updating the original symmetric binary B-trees can be done with a constant number of rotations, i.e., by constantly often rearranging subtrees. For our variant of B-trees, this implies that only a constant number of internal nodes are involved in an update procedure. In particular, if a subtree is removed or appended to a certain node, only the values of this node and of its predecessors need to be updated. The number of predecessors is bounded by the height of the tree which is logarithmic in the number of its leaves. Hence, the additional values stored in internal nodes can be maintained in time $\mathcal{O}(\log n)$ or $\mathcal{O}(\log m)$. Storing the additional values such as total size of a subtree in its root allows us to compute prefixes or the prefix sum with respect to these values in time $\mathcal{O}(\log n)$ or $\mathcal{O}(\log m)$. *Prefix computation* refers to finding the maximal prefix of the sorted list such that the elements belonging to the prefix have values that are or whose sum is bounded by a given input. We return a prefix by outputting the index of its last element.

Lemma 7.2. *There is a data structure storing n' elements sorted with respect to a key value. Insertion, deletion, or search by key value or index of an element takes time $\mathcal{O}(\log n')$, and prefixes and prefix sums with respect to additionally stored values can be computed in time $\mathcal{O}(\log n')$.*

Hardness of Computation To conclude this section, we provide a justification for the different running times of our algorithms for Multiple Knapsack depending on the number of knapsacks. It is known that Multiple Knapsack with two identical knapsacks does not admit an FPTAS, unless $\mathcal{P} = \mathcal{NP}$ [CK05].

We are able to extend this result to the case where $m < \frac{1}{3\varepsilon}$. More precisely, we show that a $(1+\varepsilon)$-approximation algorithm for Multiple Knapsack with $m < \frac{1}{3\varepsilon}$ and running time polynomial in n and $\frac{1}{\varepsilon}$ would imply that 3-Partition can be decided in polynomial time. For the dynamic setting, this implies that there is no dynamic algorithm with update time polynomial in n and $\frac{1}{\varepsilon}$, unless $\mathcal{P} = \mathcal{NP}$. We note that this result can be extended to a larger number knapsacks with arbitrary capacities by adding an appropriate number of small knapsacks that cannot be used to pack any item.

Theorem 7.3. *Unless $\mathcal{P} = \mathcal{NP}$, there is no algorithm for* Multiple Knapsack *that maintains a $(1+\varepsilon)$-approximate solution in update time polynomial in n and $\frac{1}{\varepsilon}$ for $m < \frac{1}{3\varepsilon}$.*

Proof. Consider the strongly $\mathcal{NP}$-hard problem 3-Partition where there are $3m$ items with sizes $a_j \in \mathbb{N}$ such that $\sum_{j=1}^{3m} a_j = mA$. The task is to decide whether there exists a partition $(P_i)_{i=1}^m$ of $[3m]$ such that $|P_i| = 3$ and $\sum_{j \in P_i} a_j = A$ for every $j \in [3m]$. We note that this problem remains strongly $\mathcal{NP}$-hard even if the item sizes a_j satisfy $\frac{A}{4} < a_j < \frac{A}{2}$ [KPP04,GJ79].

Consider the following instance of Multiple Knapsack: There are m knapsacks with capacity $S = A$ and $3m$ items. Each item corresponds to a 3-Partition item with $s_j = a_j$ and $v_j = 1$ for $j \in [3m]$. Observe that the 3-Partition instance is a Yes-instance if and only if the optimal solution to the Knapsack problem contains $3m$ items.

If Multiple Knapsack admits an algorithm with approximation factor $(1+\varepsilon)$ and running time polynomial in $\frac{1}{\varepsilon}$ and n where $m < \frac{1}{3\varepsilon}$, such an algorithm is able to optimally solve the Knapsack instance reduced from 3-Partition. Therefore, such an algorithm decides 3-Partition in polynomial time which is not possible, unless $\mathcal{P} = \mathcal{NP}$. □

7.3 Dynamic Linear Grouping

We describe and analyze our dynamic approach to linear grouping for an item set $\mathcal{J}' \subseteq \mathcal{J}$ and a number $n' \leq |\mathcal{J}'|$ such that *any* optimal solution can pack at most n' items of $\mathcal{J}'$. We consider $\mathcal{J}'$ instead of $\mathcal{J}$ because one of our dynamic algorithms only uses dynamic linear grouping on a subset of items. The aim of linear grouping (and of our dynamic version) is to transform the items into *item types* of identical size and value to simplify the computation and achieve the desired update times.

Theorem 7.4. *Given a set $\mathcal{J}'$ with $|\textsc{Opt} \cap \mathcal{J}'| \leq n'$ for all optimal solutions* Opt*, there is an algorithm with running time $\mathcal{O}\left(\frac{\log^4 n'}{\varepsilon^4}\right)$ that transforms the items in $\mathcal{J}'$ into $\mathcal{O}\left(\frac{\log^2 n'}{\varepsilon^4}\right)$ item types $\mathcal{T}$ and ensures $v(\textsc{Opt}_{\mathcal{T}}) \geq \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2} v(\textsc{Opt})$. Here, $\textsc{Opt}_{\mathcal{T}}$ is an optimal solution for the modified instance induced by the item types $\mathcal{T}$ and their multiplicities and the items $\mathcal{J} \setminus \mathcal{J}'$.*

7.3.1 Algorithm

We now describe the algorithm that we use for proving Theorem 7.4. In the following, we use the notation X' for a set X to refer to $X \cap \mathcal{J}'$ while X'' refers to $X \setminus \mathcal{J}'$. Further, we fix an optimal solution Opt. Recall that, upon arrival, item values of items in $\mathcal{J}$ are rounded to natural powers of $(1+\varepsilon)$ to create the value classes V_ℓ, where each item $j \in V_\ell$ is of value $(1+\varepsilon)^\ell$.

The idea of the algorithm is based on the following observation: Knowing $n_\ell := |\text{Opt} \cap V_\ell'|$ is sufficient to determine the exact subset of V_ℓ' packed in Opt since, without loss of generality, the *smallest* n_ℓ items are packed. Given n_ℓ, the classical linear grouping approach developed by de la Vega and Lueker [dlVL81] could be applied to create item groups that ultimately reduce the number of different items. However, in a dynamic context, computing (or even guessing) n_ℓ is intractable. Hence, the algorithm creates item groups *simultaneously* for various guesses of n_ℓ before rounding the item sizes according to linear grouping. Illustrations of linear grouping and dynamic linear grouping are shown in Figures 7.1 and 7.2, respectively.

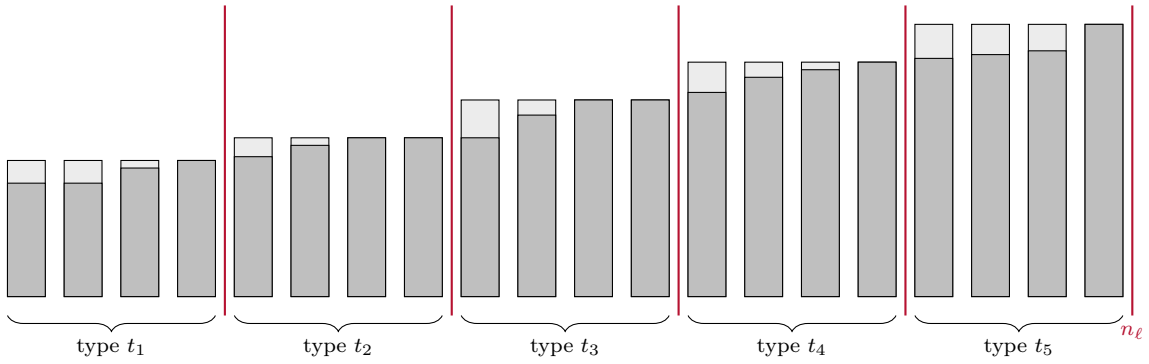

Figure 7.1: Linear grouping for items in V_ℓ' given n_ℓ. Dark rectangles correspond to the original item sizes and light rectangles indicate the rounding to item types.

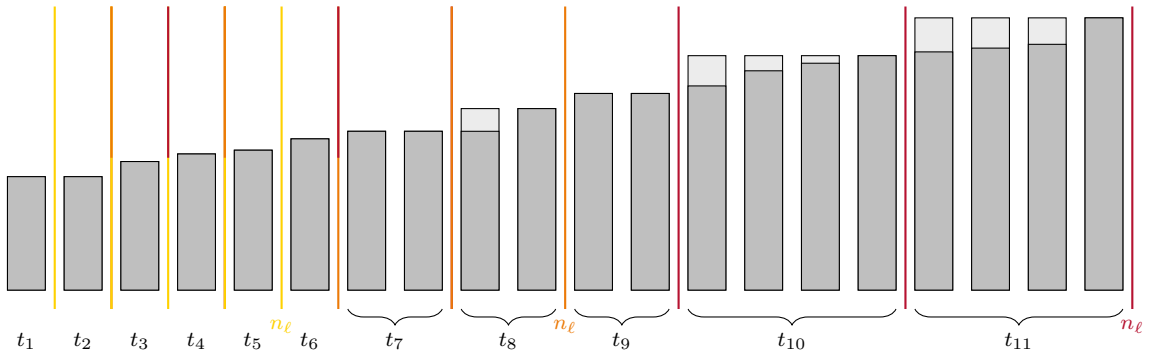

Figure 7.2: Dynamic linear grouping for items in V_ℓ'. Each color corresponds to one guess of n_ℓ.

The algorithm assumes that $\ell_{\max}$, the largest index of a value class with $V_\ell' \cap \text{Opt} \neq \emptyset$, is given as input. Next, we set $\ell_{\min} := \ell_{\max} - \left\lceil \log_{1+\varepsilon} \frac{n'}{\varepsilon} \right\rceil$. Each item $j \in V_\ell'$ for $\ell \notin \{\ell_{\min}, \dots, \ell_{\max}\}$ is discarded. For each value class $\ell \in \{\ell_{\min}, \dots, \ell_{\max}\}$ and each $l \in \{0, \dots, \lfloor \log_{1+\varepsilon} n \rfloor\}$, we

consider $\lceil (1+\varepsilon)^l \rceil$ as guess for n_ℓ and do the following. We determine the first n_ℓ items of V'_ℓ (sorted by non-decreasing size) and create $\frac{1}{\varepsilon}$ almost equal-size groups $G_1(n_\ell), \dots, G_{1/\varepsilon}(n_\ell)$. Group $G_1(n_\ell)$ contains the $\lfloor \varepsilon n_\ell \rfloor$ smallest items in V'_ℓ, and, for general $k \in \left[\frac{1}{\varepsilon}\right]$, $G_k(n_\ell)$ contains the $\lfloor \varepsilon n_\ell \rfloor$ or $\lceil \varepsilon n_\ell \rceil$ smallest items in V'_ℓ not contained in $G_{k'}(n_\ell)$ for $k' < k$. If $\varepsilon n_\ell \notin \mathbb{N}$, we ensure that $|G_k(n_\ell)| \leq |G_{k'}(n_\ell)|$ for $k \leq k'$. If $\frac{1}{\varepsilon}$ was not yet considered as guess for n_ℓ, then we also create $G_1\left(\frac{1}{\varepsilon}\right), \dots, G_{1/\varepsilon}\left(\frac{1}{\varepsilon}\right)$, where $G_k\left(\frac{1}{\varepsilon}\right)$ contains the kth smallest item in V'_ℓ.

For one guess of n_ℓ, let $j_k(n_\ell)$ be the last job in V'_ℓ belonging to group $G_k(n_\ell)$. After having determined $j_k(n_\ell)$ for each possible value n_ℓ $\left(\text{including } \frac{1}{\varepsilon}\right)$ and for each $k \in \left[\frac{1}{\varepsilon}\right]$, the size of each item $j \in V'_\ell$ is rounded up to the size of the next larger item $j^\star$ with $j^\star = j_k(n_\ell)$ for some combination of k and n_ℓ. That is, each item belongs to an item type t with size s_t and value v_t. We summarize the algorithm in Algorithm 7.1. Without loss of generality, we use the position of an item $j \in V'_\ell$, when V_ℓ is sorted by non-decreasing size, to refer to the item itself.

Algorithm 7.1: Dynamic linear grouping

$\ell_{\max} \leftarrow$ guess of the largest index of a value class with $V'_\ell \cap \textsc{Opt} \neq \emptyset$
$\ell_{\min} \leftarrow \ell_{\max} - \lceil \log_{1+\varepsilon} \frac{n'}{\varepsilon} \rceil$
for $\ell = \ell_{\min}, \dots, \ell_{\max}$ do
    for $l = 0, \dots, \lfloor \log_{1+\varepsilon} n' \rfloor$ do
        $n_\ell \leftarrow \lceil (1+\varepsilon)^l \rceil$
        for $k = 1, \dots, \frac{1}{\varepsilon}$ do
            determine $G_k(n_\ell)$ and $j_k(n_\ell) \leftarrow \max\{j : j \in G_k(n_\ell)\}$
    if $\frac{1}{\varepsilon} \neq \lceil (1+\varepsilon)^l \rceil$ for some $l \in \{0, \dots, \lfloor \log_{1+\varepsilon} n' \rfloor\}$ do
        determine $G_1(\frac{1}{\varepsilon}), \dots, G_{1/\varepsilon}(\frac{1}{\varepsilon})$
        for $k = 1, \dots, \frac{1}{\varepsilon}$ do
            $j_k(\frac{1}{\varepsilon}) \leftarrow \max\{j : j \in G_k(\frac{1}{\varepsilon})\}$
    for $j \in V'_\ell$ do
        $j^\star \leftarrow \min_{k,n_\ell}\{j_k(n_\ell) : j_k(n_\ell) \geq j\}$
        $\tilde{s}_j \leftarrow s_{j^\star}$
for $\ell < \ell_{\min}$ and $\ell > \ell_{\max}$ do
    discard each item $j \in V'_\ell$

7.3.2 Analysis

We start by observing that the loss in the objective function due to rounding item values to natural powers of $(1+\varepsilon)$ is bounded by a factor of $\frac{1}{1+\varepsilon}$; see Lemma 7.1. Let $V_{\ell_{\max}}$ be the highest value class with $V'_\ell \cap \textsc{Opt} \neq \emptyset$. As $\ell_{\min}$ is chosen such that n' items of value at most $(1+\varepsilon)^{\ell_{\min}}$ contribute less than an ε-fraction to $v(\textsc{Opt})$, the loss in the objective function by discarding items in value classes V'_ℓ with $\ell \notin \{\ell_{\min}, \dots, \ell_{\max}\}$ is bounded by a factor $(1-\varepsilon)$ as we show in Lemma 7.6. By taking only $\left\lceil (1+\varepsilon)^{\lfloor \log_{1+\varepsilon} n_\ell \rfloor} \right\rceil$ items of V'_ℓ instead of n_ℓ, we lose

at most a factor of $\frac{1}{1+\varepsilon}$; see Lemma 7.7. Observing that the groups created by dynamic linear grouping are an actual refinement of the groups created by the classical linear grouping for a fixed number of items, we pack our items as done in linear grouping: Not packing the group with the largest items allows us to "shift" all rounded items of group $G_k(n_\ell)$ to the positions of the (not rounded) items in group $G_{k+1}(n_\ell)$ at the expense of losing a factor of $(1-2\varepsilon)$ as we see in Lemma 7.8. Combining these results then shows the following lemma.

Lemma 7.5. *There is an index $\ell_{\max}$ such that $v(\textsc{Opt}_{\mathcal{T}}) \geq \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2} v(\textsc{Opt})$, where $\textsc{Opt}_{\mathcal{T}}$ is an optimal packing for the modified instance induced by the item types $\mathcal{T}$ and their multiplicities and the set $\mathcal{J}''$.*

Let $\mathcal{P}_1$ be the set of solutions that may use all items in $\mathcal{J}''$ and uses items in $\mathcal{J}'$ only of the value classes V_ℓ with $\ell \in \{\ell_{\min}, \dots, \ell_{\max}\}$. Let $\textsc{Opt}_1$ be an optimal solution in $\mathcal{P}_1$. The following lemma bounds the value of $\textsc{Opt}_1$ in terms of $\textsc{Opt}$.

Lemma 7.6. $v(\textsc{Opt}_1) \geq (1-\varepsilon) v(\textsc{Opt})$.

Proof. From being given $\ell_{\max}$, we know that $v(\textsc{Opt}) \geq (1+\varepsilon)^{\ell_{\max}}$. As n' is an upper bound on the cardinality of $\textsc{Opt}'$, the items in the value classes V_ℓ with $\ell < \ell_{\min}$ contribute at most $n'-1$ items to $\textsc{Opt}'$ while the value of one such item is bounded by $(1+\varepsilon)^{\ell_{\min}}$. Thus, the total value of items in $V_0, \dots, V_{\ell_{\min}-1}$ contributing to $\textsc{Opt}'$ is bounded by

$$n'(1+\varepsilon)^{\ell_{\min}} = n'(1+\varepsilon)^{\ell_{\max} - \lceil \log_{1+\varepsilon} n'/\varepsilon \rceil} \leq \varepsilon (1+\varepsilon)^{\ell_{\max}} \leq \varepsilon v(\textsc{Opt}),$$

while the items in V_ℓ with $\ell > \ell_{\max}$ do not contribute to $v(\textsc{Opt})$.

Let $\mathcal{J}_1$ be the set of items in $\textsc{Opt}'$ restricted to the value classes V_ℓ with $\ell \in \{\ell_{\min}, \dots, \ell_{\max}\}$. Clearly, $\mathcal{J}_1$ and $\textsc{Opt}''$ together can be feasibly packed. Hence,

$$v(\textsc{Opt}_1) \geq v(\mathcal{J}_1) + v(\textsc{Opt}'') \geq v(\textsc{Opt}') - \varepsilon v(\textsc{Opt}) + v(\textsc{Opt}'') \geq (1-\varepsilon) v(\textsc{Opt}),$$

which concludes the proof. □

From now on, we only consider packings in $\mathcal{P}_1$, i.e., we restrict $\mathcal{J}'$ to items in the value classes V_ℓ with $\ell \in \{\ell_{\min}, \dots, \ell_{\max}\}$. Let V_ℓ be a value class contributing to $\textsc{Opt}'_1$. As explained above, knowing $n_\ell = |V'_\ell \cap \textsc{Opt}_1|$ would be sufficient to determine the items of V'_ℓ in $\textsc{Opt}_1$, i.e., to determine $V'_\ell \cap \textsc{Opt}_1$. In the following lemma, we show that we can additionally assume that $n_\ell = 0$ or $n_\ell = \lceil (1+\varepsilon)^{k_\ell} \rceil$ for some $k_\ell \in \mathbb{N}_0$. To this end, let $\mathcal{P}_2$ contain all the packings in $\mathcal{P}_1$ where the number of big items of each value class V_ℓ is either 0 or $\lceil (1+\varepsilon)^{k_\ell} \rceil$ for some $k_\ell \in \mathbb{N}_0$. Let $\textsc{Opt}_2$ be an optimal packing in $\mathcal{P}_2$.

Lemma 7.7. $v(\textsc{Opt}_2) \geq \frac{1}{(1+\varepsilon)} v(\textsc{Opt}_1)$.

Proof. Consider OPT_1, an optimal packing in $\mathcal{P}_1$. We construct a feasible packing in $\mathcal{P}_2$ that achieves the desired value of $\frac{1}{(1+\varepsilon)}v(\text{OPT}_1)$. Let $\mathcal{J}_2$ be the subset of OPT_1' where each value class V_ℓ' is restricted to the smallest $\left\lceil (1+\varepsilon)^{\lfloor \log_{1+\varepsilon} n_\ell \rfloor} \right\rceil$ items in V_ℓ' if $V_\ell' \cap \text{OPT}_1 \neq \emptyset$.

Fix one value class V_ℓ with $V_\ell' \cap \text{OPT}_1 \neq \emptyset$. Restricting to the first $\left\lceil (1+\varepsilon)^{\lfloor \log_{1+\varepsilon} n_\ell \rfloor} \right\rceil$ items in $V_\ell \cap \text{OPT}_1'$ implies

$$v(V_\ell \cap \mathcal{J}_2) \geq (1+\varepsilon)^{\lfloor \log_{1+\varepsilon} n_\ell \rfloor}(1+\varepsilon)^\ell \geq \frac{1}{1+\varepsilon}(1+\varepsilon)^\ell n_\ell = \frac{1}{1+\varepsilon}v(V_\ell' \cap \text{OPT}_1).$$

Clearly, $\mathcal{J}_2 \cup \text{OPT}_1''$ is a feasible packing in $\mathcal{P}_2$. Since $v(\text{OPT}_1') = \sum_{\ell=\ell_{\min}}^{\ell_{\max}} v(V_\ell' \cap \text{OPT}_1)$,

$$v(\text{OPT}_2) \geq v(\mathcal{J}_2) + v(\text{OPT}_1'') \geq \frac{1}{1+\varepsilon}v(\text{OPT}_1') + v(\text{OPT}_1'') \geq \frac{1}{1+\varepsilon}v(\text{OPT}_1).$$

This proves the statement of the lemma. □

From now on, we only consider packings in $\mathcal{P}_2$. This means, we restrict the items in $\mathcal{J}'$ to value classes V_ℓ' with $\ell \in \{\ell_{\min}, \dots, \ell_{\max}\}$ and assume that $n_\ell = \lceil (1+\varepsilon)^{k_\ell} \rceil$ for $k_\ell \in \mathbb{N}_0$ or $n_\ell = 0$. Even with n_ℓ being of the form $\lceil (1+\varepsilon)^{k_\ell} \rceil$, independently guessing the exponent for each value class V_ℓ' is infeasible in time polynomial in $\log n$ and $\frac{1}{\varepsilon}$. To resolve this, the dynamic linear grouping creates groups that take into account all possible guesses of n_ℓ. The dynamic linear grouping results in item types $\mathcal{T}_\ell$ and their multiplicities for the set V_ℓ'.

Let $\mathcal{P}_\mathcal{T}$ be the set of all feasible packings for the modified instance induced by the item types $\mathcal{T}_\ell$ for $\ell_{\min} \leq \ell \leq \ell_{\max}$ and their multiplicities and the set $\mathcal{J}''$. That is, instead of the original items in $\mathcal{J}'$, the packings in $\mathcal{P}_\mathcal{T}$ pack the corresponding item types. Note that packings in $\mathcal{P}_\mathcal{T}$ are not forced to pack a specific number of items per value class. Let $\text{OPT}_\mathcal{T}$ be an optimal solution in $\mathcal{P}_\mathcal{T}$. The next lemma shows that $v(\text{OPT}_\mathcal{T})$ is at most a factor $(1-2\varepsilon)$ less than $v(\text{OPT}_2)$, the optimal solution value of packings in $\mathcal{P}_2$.

Lemma 7.8. $v(\text{OPT}_\mathcal{T}) \geq (1-2\varepsilon)v(\text{OPT}_2)$.

Proof. We construct a feasible packing in $\mathcal{P}_\mathcal{T}$ based on the optimal packing OPT_2. The items in OPT_2'' are packed exactly in the same way as they are packed in OPT_2. For items in $\mathcal{J}'$, we individually consider each value class $V_\ell' \cap \text{OPT}_2$ with $\ell \in \{\ell_{\min}, \dots, \ell_{\max}\}$ and construct a set $\mathcal{J}_\ell \subseteq (V_\ell' \cap \text{OPT}_2)$ to obtain $\mathcal{J}_\mathcal{T} := \bigcup_{\ell=\ell_{\min}}^{\ell_{\max}} \mathcal{J}_\ell$. Our packing in $\mathcal{P}_\mathcal{T}$ corresponds then to the items in $\text{OPT}_2'' \cup \mathcal{J}_\mathcal{T}$. We show that the items in $\mathcal{J}_\ell$ can be packed into the space of the knapsacks where the items in $V_\ell' \cap \text{OPT}_2$ are placed while ensuring that $v(\mathcal{J}_\ell) \geq (1-2\varepsilon)v(V_\ell \cap \text{OPT}_2')$.

If $V_\ell' \cap \text{OPT}_2 = \emptyset$, then we set $\mathcal{J}_\ell = \emptyset$ and both requirements are trivially satisfied.

If $|V_\ell' \cap \text{OPT}_2| \leq \frac{1}{\varepsilon}$, we set $\mathcal{J}_\ell = V_\ell' \cap \text{OPT}_2$. Clearly, $v(\mathcal{J}_\ell) \geq (1-2\varepsilon)v(V_\ell \cap \text{OPT}_2')$. For packing $\mathcal{J}_\ell$, we observe that $\mathcal{T}_\ell$ contains the smallest $\frac{1}{\varepsilon}$ items as item types. Hence, their sizes are not affected by the rounding procedure and we can pack theses items as is done by OPT_2.

Let ℓ be a value class with $n_\ell = |V_\ell' \cap \text{OPT}_2| > \frac{1}{\varepsilon}$. Let $G_1(n_\ell), \dots, G_{1/\varepsilon}(n_\ell)$ be the corresponding $\frac{1}{\varepsilon}$ groups of $\lfloor \varepsilon n_\ell \rfloor$ or $\lceil \varepsilon n_\ell \rceil$ many items created by dynamic linear grouping. We

set $\mathcal{J}_\ell = G_1(n_\ell) \cup \ldots \cup G_{1/\varepsilon-1}(n_\ell)$. Since $v(G_{1/\varepsilon}(n_\ell)) = \lceil \varepsilon n_\ell \rceil (1+\varepsilon)^\ell \leq 2\varepsilon n_\ell (1+\varepsilon)^\ell$, we have $v(\mathcal{J}_\ell) \geq (1-2\varepsilon) v(V'_\ell \cap \text{OPT}_2)$. For packing the items in $\mathcal{J}_\ell$, we observe that the item types created by our algorithm are a refinement of $G_1(n_\ell), \ldots, G_{1/\varepsilon}(n_\ell)$. Since the dynamic linear grouping ensures that $|G_{1/\varepsilon}(n_\ell)| \geq \cdots \geq |G_1(n_\ell)|$, for $k \in \left[\frac{1}{\varepsilon}\right]$, we can pack the items of group $G_k(n_\ell)$ where OPT_2 packs the items of group $G_{k+1}(n_\ell)$. Therefore,

$$v(\text{OPT}_\mathcal{T}) \geq v(\mathcal{J}_\mathcal{T}) + v(\text{OPT}''_2) \geq (1-2\varepsilon) v(\text{OPT}'_2) + v(\text{OPT}''_2) \geq (1-2\varepsilon) v(\text{OPT}_2)$$

which concludes the proof. □

Since $\mathcal{T}$ contains at most $\frac{1}{\varepsilon}\left(\left\lceil \frac{\log n'/\varepsilon}{\log(1+\varepsilon)} \right\rceil + 1\right)$ many different value classes and since using $\left\lceil \frac{\log n'}{\log(1+\varepsilon)} \right\rceil + 1$ different values for $n_\ell = |\text{OPT} \cap V'_\ell|$ suffices as explained above, the next lemma follows.

Lemma 7.9. *The algorithm reduces the number of item types to* $\mathcal{O}\left(\frac{\log^2 n'}{\varepsilon^4}\right)$.

Lemma 7.10. *For a given guess* $\ell_{\max}$*, the set* $\mathcal{T}$ *can be computed in time* $\mathcal{O}\left(\frac{\log^4 n'}{\varepsilon^4}\right)$.

Proof. Recall that n' is an upper bound on the number of items in $\mathcal{J}'$ in any feasible solution. Observe that the boundaries of the linear grouping created by the algorithm per value class are actually independent of the value class and only refer to the kth item in some class V_ℓ. Hence, the algorithm first computes the different indices needed in this round. We denote the set of these indices by $I' = \{j_1, \ldots\}$ sorted in an increasing manner. There are at most $\lfloor \log_{1+\varepsilon} n' \rfloor$ many possibilities for n_ℓ. Thus, the algorithm needs to compute at most $\frac{1}{\varepsilon}(\log_{1+\varepsilon} n' + 1)$ many different indices. This means that these indices can be computed and stored in time $\mathcal{O}\left(\frac{\log n'}{\varepsilon^2}\right)$.

Given the guess $\ell_{\max}$ and $\ell_{\min}$, fix a value class V_ℓ with $\ell \in \{\ell_{\min}, \ldots, \ell_{\max}\}$. We want to bound the time the algorithm needs to transform the big items in V_ℓ into the modified item set $\mathcal{T}_\ell$. We will ensure that the dynamic algorithms in the following sections maintain a balanced binary search tree for each value class V_ℓ that stores the items in $\mathcal{J}'$ sorted by increasing size. Hence, the sizes of the items corresponding to I' can be accessed in time $\mathcal{O}\left(\frac{\log^3 n'}{\varepsilon^2}\right)$ to extract the item-type size s_t for $t \in \mathcal{T}_\ell$. Given an item type $t \in \mathcal{T}_\ell$, its multiplicity n_t can again be pre-computed independently of the value class. Thus, $\mathcal{T}_\ell$ can be computed in time $\mathcal{O}\left(\frac{\log^3 n'}{\varepsilon^2}\right)$.

As there are $\mathcal{O}\left(\frac{\log n'}{\varepsilon^2}\right)$ many value classes that need to be considered for a given guess $\ell_{\max}$, calculating the set $\mathcal{T}$ needs $\mathcal{O}\left(\frac{\log^4 n'}{\varepsilon^4}\right)$ many computational steps. □

Proof of Theorem 7.4. Lemma 7.5 bounds the loss in the objective function, Lemma 7.9 bounds the number of item types, and Lemma 7.10 bounds the running time. □

7.4 Identical Knapsacks

We give a dynamic algorithm that achieves an approximation ratio of $(1+\varepsilon)$ for Multiple Knapsack with identical knapsack sizes, i.e., $S_i = S$ for all $i \in [m]$. The running time of the update operation is always polynomial in $\log n$ and $\frac{1}{\varepsilon}$. In this section, we assume $m < n$ as otherwise assigning the items in some consistent order to the knapsacks is optimal. We focus on instances where m is large, i.e., $m \geq \frac{16}{\varepsilon^7}\log^2 n$ but still dynamic. For $m < \frac{16}{\varepsilon^7}\log^2 n$, we use the algorithm for few knapsacks we present in [BEM+20].

Theorem 7.11. *Let $\varepsilon > 0$ and let $U = \max\{Sm, nv_{\max}\}$. If $m \geq \frac{16}{\varepsilon^7}\log^2 n$, there is a dynamic $(1+\varepsilon)$-approximate algorithm for the* Multiple Knapsack *problem with m identical knapsacks with update time $\left(\frac{\log U}{\varepsilon}\right)^{\mathcal{O}(1)}$. Queries for single items and the solution value can be answered in time $\mathcal{O}\left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1)}$ and $\mathcal{O}(1)$, respectively. The current solution P can be returned in time $|P|\left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1)}$.*

7.4.1 Algorithm

Definitions and data structures We partition the items into two sets, $\mathcal{J}_B$, the set of *big* items, and $\mathcal{J}_S$, the set of *small* items, with sizes $s_j \geq \varepsilon S$ and $s_j < \varepsilon S$, respectively. For an optimal solution Opt, define $\text{Opt}_B = \text{Opt} \cap \mathcal{J}_B$ and $\text{Opt}_S = \text{Opt} \cap \mathcal{J}_S$.

For this algorithm, we maintain three types of data structures. We store all items in one balanced binary tree in order of their arrivals, i.e., their indices. In this tree, we store the size s_j and the value v_j of each item j and additionally store the index ℓ_j of its value class for big items. Big items are also stored in one balanced binary tree per value class V_ℓ sorted by non-decreasing size while all small items are sorted by non-increasing density and stored in one tree. Overall, we have at most $2 + \log_{1+\varepsilon} v_{\max}$ many data structures to maintain. Upon arrival of a new item, we insert it into the tree of all items and classify this item as big or small depending on whether $s_j \geq \varepsilon S$ or $s_j < \varepsilon S$. If the item is small, we insert it into the data structure for small items. Otherwise, we determine the index of its value class ℓ_j and insert it into the corresponding data structure. If the number m of knapsacks changes, we take this into account by updating the parameter m in the algorithm.

Algorithm The high-level idea of the algorithm is to apply the dynamic linear grouping approach developed in the previous section to big items. Given the thus significantly decreased number of different item types, we set up an ILP to assign big items via configurations while small items are only assigned via a (fractional) placeholder item.

More precisely, we guess the index $\ell_{\max}$ of the highest value class that belongs to Opt_B by testing each possible value $\ell_{\max} \in \{0, \dots, \lfloor \log_{1+\varepsilon} v_{\max} \rfloor\}$. Then, we use dynamic linear grouping (Algorithm 7.1) with $\mathcal{J}' = \mathcal{J}_B$ and $n' = \min\left\{\frac{m}{\varepsilon}, |\mathcal{J}_B|\right\}$ to obtain $\mathcal{T}$, the set of item types t with their multiplicities n_t.

Given these item types, we create the set of all *configurations* $\mathcal{C}$ of big items. A configuration consists of at most n_t items of type $t \in \mathcal{T}$ and is such that its total size does not exceed the knapsack capacity S. Hence, a configuration contains at most $\frac{1}{\varepsilon}$ big items. For $c \in \mathcal{C}$ and $t \in \mathcal{T}$ let $n_{c,t}$ denote the number of items of type t in configuration c. Let $v_c = \sum_{t \in \mathcal{T}} n_{c,t} v_t$ and $s_c = \sum_{t \in \mathcal{T}} n_{c,t} s_t$ denote the total value and size, respectively, of the items in c.

Next, we guess v_S, the value of OPT_S, up to a power of $(1+\varepsilon)$. Let P_S be the maximal prefix of small items with $v(P_S) < v_S$ and set $s_S = s(P_S)$. We solve the following configuration ILP with variables y_c, for $c \in \mathcal{C}$, for the current guesses $\ell_{\max}$ and s_S. Here, y_c counts how often a certain configuration c is used.

$$
\begin{aligned}
\text{max} \quad & \sum_{c \in \mathcal{C}} y_c v_c & & & \\
\text{subject to} \quad & \sum_{c \in \mathcal{C}} y_c s_c & \leq & \lfloor (1-3\varepsilon)m \rfloor S - s_S & \\
& \sum_{c \in \mathcal{C}} y_c & \leq & \lfloor (1-3\varepsilon)m \rfloor & \qquad \text{(P)} \\
& \sum_{c \in \mathcal{C}} y_c n_{c,t} & \leq & n_t & \text{for all } t \in \mathcal{T} \\
& y_c & \in & \mathbb{Z}_{\geq 0} & \text{for all } c \in \mathcal{C}
\end{aligned}
$$

The first and second inequality ensure that the configurations chosen by the ILP can be packed into $\lfloor (1-3\varepsilon)m \rfloor$ knapsacks while reserving sufficient space for the small items. The third inequality guarantees that only available items are used.

Clearly, we cannot solve the configuration ILP to optimality. Hence, we relax the integrality constraint and allow fractional solutions. Given such a fractional solution, we round it to an integral packing P_B using at most $\lfloor \varepsilon m \rfloor$ additional knapsacks while ensuring that $v(P_B) \geq v_{\text{LP}}$, where v_{LP} is the optimal solution value for the LP relaxation.

Given an integral packing of the big items, it remains to pack the small items. Let P_S be the maximal prefix of small items with $v(P_S) < v_S$ and let $j^\star$ be the densest small item not in P_S. We pack $j^\star$ into one of the knapsacks kept empty by P_B. Then, we fractionally fill up the $\lfloor (1-2\varepsilon)m \rfloor$ knapsacks used by P_B and place any item that is *cut*, i.e., placed into more than one knapsack, into the $\lceil \varepsilon m \rceil$ additional knapsacks that are still empty. We can guarantee that this packing is feasible and packs all items in $P_S \cup \{j^\star\}$.

We return the solution for the guesses $\ell_{\max}$ and v_S that maximize the total attained value. We note that the explicit packing of the items is only determined upon query. A possible (implicit) solution is shown in Figure 7.3. We summarize the algorithm in Algorithm 7.2.

Algorithm 7.2: Dynamic algorithm for identical knapsacks

guess $\ell_{\max}$, the largest index of a value class with big items in OPT
  use dynamic linear grouping for the big items to obtain $\mathcal{T}$
  guess v_S, the value of small items
    $P_S \leftarrow$ maximal prefix of small items with $v(P_S) \leq v_S$

$s_S \leftarrow s(P_S)$
$j^\star \leftarrow$ densest small item not in P_S
solve (P) for s_S and $\mathcal{T}$
use a Next−Fit Algorithm to pack the small items $P_S \cup \{j^\star\}$

We remark that we simplified the algorithm for conciseness as follows: Even after applying dynamic linear grouping to the big items, the number of feasible configurations is still prohibitively large to directly solve it. Hence, instead of creating all configurations and solving the LP relaxation of the configuration ILP, we use the Ellipsoid Method on the dual LP to determine the important configurations and reduce the number of relevant variables. As we will show, this reduces the number of configurations to a manageable amount, which enables us to solve the LP relaxation in time polynomial in $\log n$ and $\frac{1}{\varepsilon}$.

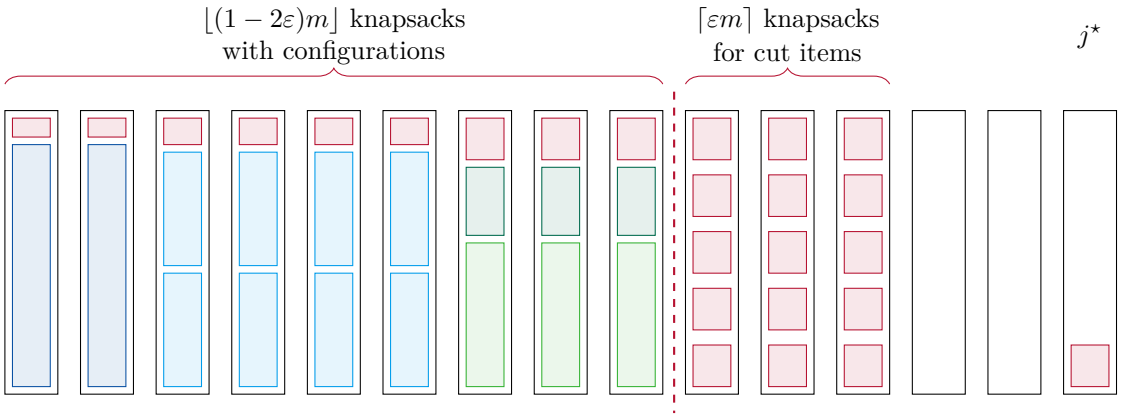

Figure 7.3: A possible solution of the Algorithm 7.2: Blue and green rectangles represent the packed big item types. Red rectangles on the left side represent the space left empty by the configurations and on the right represent the slots for cut items.

Queries We explain how to efficiently answer different queries. Instead of explicitly storing the packing of any item, we define and update pointers for small items and for each item type that dictate the knapsack where the next queried item of the respective type is packed. To stay consistent for the precise packing of a particular item between two update operations, we additionally cache query answers for the current round in the data structure that stores items. We give the technical details in the next section.

- **Single Item Query:** If the queried item is small, we check if it belongs to the prefix of densest items that is part of our solution. In this case, the pointer for small items determines the knapsack. If the queried item is big, we retrieve its item type and check if it belongs to smallest items of this type that are packed by the implicit solution. In this case, the pointer for this item type dictates the knapsack.

- **Solution Value Query:** As the algorithm works with rounded values, after having found the current solution, we use prefix computation on the small items and on any value class of big items to calculate and store the actual solution value. When queried, we return the stored solution value in constant time.

- **Entire Solution Query:** We use prefix computation on the small items as well as on the value classes of the big items to retrieve the packed items. Next, we use the single item query to determine their respective knapsacks.

7.4.2 Analysis

Setting up the configuration ILP The first step is to analyze the loss in the objective function value due to the linear grouping. To this end, set $\mathcal{J}' = \mathcal{J}_B$ and $n' = \min\{\frac{m}{\varepsilon}, |\mathcal{J}_B|\}$. Moreover, let $\text{OPT}_{\mathcal{T}}$ be an optimal packing for the instance induced by the item types $\mathcal{T}$ (obtained from applying dynamic linear grouping to $\mathcal{J}_B$) and their multiplicities as well as $\mathcal{J}_S$. Then, the next corollary immediately follows from Theorem 7.4.

Corollary 7.12. *There exists an index $\ell_{\max}$ such that $v(\text{OPT}_{\mathcal{T}}) \geq \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2} v(\text{OPT})$.*

In the next lemma, we show that there is a guess v_S with corresponding size s_S such that $v_{\text{ILP}} + v(P_S) + v_{j^\star}$, with the optimal solution value v_{ILP} of (P), is a good guess for the optimal solution value $v(\text{OPT}_{\mathcal{T}})$. Here, $j^\star$ is the densest small item not contained in P_S, and P_S is the maximal prefix of small items with $v(P_S) < v_S$. The high-level idea of the proof is to restrict an optimal solution $\text{OPT}_{\mathcal{T}}$ to $\lfloor(1-3\varepsilon)m\rfloor$ most valuable knapsacks and to show that s_S underestimates the size of small items in these $\lfloor(1-3\varepsilon)m\rfloor$ knapsacks. Transforming these knapsacks into configurations yields a feasible solution for the configuration ILP.

Lemma 7.13. *There is a guess v_S with $v_{ILP} + v_S \geq \frac{1-4\varepsilon}{1+\varepsilon} v(\text{OPT}_{\mathcal{T}})$. Moreover, $v(P_S) + v_{j^\star} \geq v_S$.*

Proof. Let $\text{OPT}_{B,\mathcal{T}} := \text{OPT}_{\mathcal{T}} \cap \mathcal{J}_B$ and $\text{OPT}_{S,\mathcal{T}} := \text{OPT}_{\mathcal{T}} \cap \mathcal{J}_S$. We construct a candidate set $\mathcal{J}_{\text{ILP}}$ of items that are feasible for (P) and obtain a value of at least $(1-4\varepsilon)v(\text{OPT}_{B,\mathcal{T}})$. To this end, take an optimal packing $\text{OPT}_{\mathcal{T}}$ and consider the $\lfloor(1-3\varepsilon)m\rfloor$ most valuable knapsacks in this packing. Let $\mathcal{J}_{B,\mathcal{T}}$ and $\mathcal{J}_{S,\mathcal{T}}$ consist of the big and small items, respectively, in these knapsacks. Since $m \geq \frac{16}{\varepsilon^7}\log^2 n$, we have $\lfloor(1-3\varepsilon)m\rfloor \geq (1-4\varepsilon)m$. Hence,

$$v(\mathcal{J}_{B,\mathcal{T}}) + v(\mathcal{J}_{S,\mathcal{T}}) \geq (1-4\varepsilon)v(\text{OPT}_{\mathcal{T}})\,.$$

Create the variable values y_c corresponding to the number of times configuration c is used by the items in $\mathcal{J}_{B,\mathcal{T}}$. We observe that $\mathcal{J}_{B,\mathcal{T}} \cup \mathcal{J}_{S,\mathcal{T}}$ can be feasibly packed into $\lfloor(1-3\varepsilon)m\rfloor$ knapsacks. Therefore,

$$\sum_{c\in\mathcal{C}} y_c \leq \lfloor(1-3\varepsilon)m\rfloor\,,$$

and

$$\sum_{c\in\mathcal{C}} y_c s_c + s(\mathcal{J}_{S,\mathcal{T}}) \leq \lfloor (1-3\varepsilon)m \rfloor S\,.$$

Since we guess the value of the small items in the dynamic algorithm up to a factor of $(1+\varepsilon)$, there is one guess v_S satisfying $v_S \leq v(\mathcal{J}_{S,\mathcal{T}}) < (1+\varepsilon)v_S$. Let P_S be the maximal prefix of small items with $v(P_S) < v_S$ and let $j^\star$ be the densest small item not in P_S. Hence,

$$v(P_S) + v_{j^\star} \geq v_S \geq \frac{1}{1+\varepsilon} v(\mathcal{J}_{S,\mathcal{T}}).$$

As P_S contains the densest small items, this implies $s_S := s(P_S) \leq s(\mathcal{J}_{S,\mathcal{T}})$. Thus,

$$\sum_{c\in\mathcal{C}} y_c s_c \leq \lfloor (1-3\varepsilon)m \rfloor S - s(\mathcal{J}_{S,\mathcal{T}}) \leq \lfloor (1-3\varepsilon)m \rfloor S - s_S.$$

Therefore, the just created y_c are feasible for the ILP with the guess v_S, and

$$v_{\text{ILP}} + v_S \geq v(\mathcal{J}_{B,\mathcal{T}}) + \frac{1}{1+\varepsilon} v(\mathcal{J}_{S,\mathcal{T}}) \geq \frac{1}{1+\varepsilon}(v(\mathcal{J}_{B,\mathcal{T}}) + v(\mathcal{J}_{S,\mathcal{T}})) \geq \frac{1-4\varepsilon}{1+\varepsilon} v(\textsc{Opt}_{\mathcal{T}}),$$

which concludes the proof. □

Solving the LP relaxation Next, we explain how to approximately solve the LP relaxation of the configuration ILP (P) and round the solution to an integral packing in slightly more knapsacks. Since any basic feasible solution of (P) has at most $\mathcal{O}(|\mathcal{T}|)$ strictly positive variables, solving its dual problem with the Grötschel-Lovasz-Schrijver [GLS81] variant of the Ellipsoid Method determines the relevant variables. We refer to the books by Bertsimas and Tsitsiklis [BT97] and Papadimitriou and Steiglitz [PS82] for details on the Ellipsoid Method.

As we will show, the separation problem is a Knapsack problem, which we can solve only approximately in time polynomial in $\log n$ and $\frac{1}{\varepsilon}$, unless $\mathcal{P} = \mathcal{NP}$. The approximate separation oracle we develop correctly detects infeasibility while a solution that is declared feasible may only be feasible for a closely related problem causing a loss in the objective function value of a factor at most $(1-\varepsilon)$. We cannot use the approaches by Plotkin, Shmoys, and Tardos [PST95] or Karmarkar and Karp [KK82] directly because our configuration ILP contains two extra constraints which correspond to additional variables in the dual and thus to two extra terms in the objective function. Instead, we add an objective function constraint to the dual and test feasibility for a set of geometrically increasing guesses of the objective function value. Given the maximal guess for which the dual is infeasible, we use the variables corresponding to constraints added by the Ellipsoid Method to solve the primal. The multiplicative gap between the maximal infeasible and the minimal feasible such guess allows us to obtain a fractional solution with objective function value at least $\frac{1-\varepsilon}{1+\varepsilon} v_{\text{LP}}$, where v_{LP} is the optimal objective function value of the LP relaxation of (P).

Lemma 7.14. *Let $U = \max\{Sm, nv_{\max}\}$. Then, there is an algorithm that finds a feasible solution for the LP relaxation of* (P) *with value at least $\frac{1-\varepsilon}{1+\varepsilon} v_{LP}$ and with running time bounded by $\left(\frac{\log U}{\varepsilon}\right)^{\mathcal{O}(1)}$.*

For proving this lemma, we abuse notation and also refer to the LP relaxation of (P) by (P):

$$\begin{array}{lllll} \max & \sum_{c\in\mathcal{C}} y_c v_c & & & \\ \text{subject to} & \sum_{c\in\mathcal{C}} y_c s_c & \leq & \lfloor(1-3\varepsilon)m\rfloor S - s_S & \\ & \sum_{c\in\mathcal{C}} y_c & \leq & \lfloor(1-3\varepsilon)m\rfloor & \\ & \sum_{c\in\mathcal{C}} y_c n_{tc} & \leq & n_t & \text{for all } t\in\mathcal{T} \\ & y_c & \geq & 0 & \text{for all } c\in\mathcal{C} \end{array} \tag{P}$$

Let γ and β be the dual variables of the capacity constraint and the number-of-knapsacks constraint, respectively. Let α_t for $t \in \mathcal{T}$ be the dual variable of the constraint ensuring that only n_t items of type t are packed. Then, the dual is given by the following linear program.

$$\begin{array}{lllll} \min & \lfloor(1-3\varepsilon)m\rfloor\beta + (\lfloor(1-3\varepsilon)m\rfloor S - s_S)\gamma + \sum_{t\in\mathcal{T}} n_t\alpha_t & & & \\ \text{subject to} & \beta + s_c\gamma + \sum_{t\in T} \alpha_t n_{tc} & \geq & v_c & \text{for all } c\in\mathcal{C} \\ & \alpha_t & \geq & 0 & \text{for all } t\in\mathcal{T} \\ & \beta,\gamma & \geq & 0 & \end{array} \tag{D}$$

As discussed above, for applying the Ellipsoid Method, we need to solve the separation problem efficiently. The separation problem either confirms that the current solution $(\alpha^*, \beta^*, \gamma^*)$ is feasible or finds a violated constraint. As we will see, verifying the first constraint of (D) corresponds to solving a Knapsack problem. Hence, we do not expect to optimally solve the separation problem in time polynomial in $\log n$ and $\frac{1}{\varepsilon}$. Instead, we apply the dynamic program (DP) for Knapsack by Lawler [Law79] after restricting the item set further and rounding the item values. This modification is necessary to obtain a sufficiently small running time.

Let $\bar{v}_t = v_t - \alpha_t^* - \gamma^* s_t$ for $t \in \mathcal{T}$. If there exists an item type with $\bar{v}_t > \beta^*$, we return the configuration using one item of this item type. Otherwise, we set $\tilde{v}_t = \left\lfloor \frac{\bar{v}_t}{\varepsilon^4\beta^*} \right\rfloor \cdot \varepsilon^4\beta^*$. By running the DP [Law79] for Knapsack for a knapsack of capacity S on the item set $\mathcal{T}$ with multiplicities $\min\left\{\frac{1}{\varepsilon}, n_t\right\}$ and values $\tilde{v}_t$, we obtain a solution x^* where x_t^* indicates how often item type t is packed. If $\sum_{t\in\mathcal{T}} x_t^*\tilde{v}_t > \beta^*$, we return the configuration defined by x^* as separating hyperplane. Otherwise, we return declared feasible for the current solution. We summarize the algorithm in Algorithm 7.3.

Algorithm 7.3: Separation oracle

for $t \in \mathcal{T}$ do

```
    v̄_t ← v_t − α*_t − γ* s_t
if v̄_{t'} > β* for some t do // separating hyperplane
    return c with n_{c,t} = 1 for t = t' and n_{c,t} = 0 otherwise
else
    for t ∈ T do
        ṽ_t ← ⌊v̄_t/(ε⁴β*)⌋ · ε⁴β*
    run DP to obtain x*
    if Σ_{t∈T} x*_t ṽ_t > β* do // separating hyperplane
        return x*
    else
        return DECLARED FEASIBLE
```

The next lemma shows that this algorithm approximately solves the separation problem by either correctly declaring infeasibility and giving a feasible separating hyperplane or by finding a solution that is almost feasible for (D). The slight infeasibility for the dual problem will translate to a small decrease in the optimal objective function value of the primal problem.

Lemma 7.15. *Given $(\alpha^*, \beta^*, \gamma^*)$, there is an algorithm with running time $\mathcal{O}\left(\frac{\log^4 n}{\varepsilon^{14}}\right)$ which either guarantees that $\beta^* + s_c\gamma^* + \sum_{t\in\mathcal{T}} \alpha_t^* n_{tc} \geq (1-\varepsilon)v_c$ holds for all $c \in \mathcal{C}$ or finds a configuration $c \in \mathcal{C}$ with $\beta^* + s_c\gamma^* + \sum_{t\in\mathcal{T}} \alpha_t^* n_{tc} < v_c$.*

Proof. Fix a configuration c and recall that $s_c = \sum_{t\in\mathcal{T}} n_{tc}s_t$ and $v_c = \sum_{t\in\mathcal{T}} n_{tc}v_t$. Then, checking $\beta^* + s_c\gamma^* + \sum_{t\in\mathcal{T}} \alpha_t^* n_{tc} \geq v_c$ for all configurations $c \in \mathcal{C}$ is equivalent to showing $\max_{c\in\mathcal{C}} \sum_{t\in\mathcal{T}} (v_t - \alpha_t^* - \gamma^* s_t)n_{tc} \leq \beta^*$. This problem translates to solving the following ILP and comparing its objective function value to β^*.

$$\begin{aligned} \max \quad & \sum_{t\in\mathcal{T}} (v_t - \alpha_t^* - \gamma^* s_t)x_t & & & \\ \text{s.t.} \quad & \sum_{t\in\mathcal{T}} s_t x_t & \leq & S & \\ & x_t & \leq & n_t \quad \text{for all } t \in \mathcal{T} & \\ & x_t & \in & \mathbb{Z}^+ & \end{aligned} \tag{S}$$

This ILP is itself a (single) KNAPSACK problem. Hence, the solution x^* found by the Algorithm 7.3 is indeed feasible for (S).

We start by bounding the running time of Algorithm 7.3. For each item type $t \in \mathcal{T}$, we have $\bar{v}_t = v_t - \alpha_t^* - \gamma^* s_t$ and $\tilde{v}_t = \left\lfloor \frac{\bar{v}_t}{\varepsilon^4\beta^*} \right\rfloor \cdot \varepsilon^4\beta^*$. Observe that $\mathcal{T}$ only contains big items. Hence, it suffices to consider $\min\left\{\frac{1}{\varepsilon}, n_t\right\}$ items per item type in the DP. It can be checked in time $\mathcal{O}\left(\frac{\log^2 n}{\varepsilon^4}\right)$, whether $\bar{v}_t \leq \beta^*$ is violated for some $t \in \mathcal{T}$. Otherwise, the running time of the DP is bounded by $\mathcal{O}\left(\frac{|\mathcal{T}|^2}{\varepsilon^6}\right) = \mathcal{O}\left(\frac{\log^4 n}{\varepsilon^{14}}\right)$ [Law79].

It remains to show that the solution x^* either defines a separating hyperplane, i.e., a configuration c with $\beta^* + s_c\gamma^* + \sum_{t\in\mathcal{T}} \alpha_t^* n_{tc} < v_c$, or ensures $\beta^* + s_c\gamma^* + \sum_{t\in\mathcal{T}} \alpha_t^* n_{tc} \geq (1-\varepsilon)v_c$

for all $c \in \mathcal{C}$. If $\sum_{t\in\mathcal{T}} x_t^* \tilde{v}_t > \beta^*$, then

$$\sum_{t\in\mathcal{T}} x_t^* \bar{v}_t \geq \sum_{t\in\mathcal{T}} x_t^* \tilde{v}_t > \beta^*,$$

and thus x^* defines a separating hyperplane.

Consider now $\sum_{t\in\mathcal{T}} x_t^* \tilde{v}_t \leq \beta^*$. Toward a contradiction, suppose that there is a configuration c', defined by packing x_t items of type t, such that

$$\sum_{t\in\mathcal{T}} x_t \Big((1-\varepsilon)v_t - \alpha_t^* - \gamma^* s_t\Big) > \beta^*.$$

Since $\mathcal{T}$ contains only big item types, we have $\sum_{t\in\mathcal{T}} x_t \leq \frac{1}{\varepsilon}$. This implies that there exists at least one item type t' in $\mathcal{T}$ with $x_{t'} \geq 1$ and $(1-\varepsilon)v_{t'} - \alpha_{t'}^* - \gamma^* s_{t'} \geq \varepsilon\beta^*$. Moreover,

$$\bar{v}_t = v_t - \alpha_t^* - \gamma^* s_t \geq (1-\varepsilon)v_t - \alpha_t^* - \gamma^* s_t$$

holds for all item types $t \in \mathcal{T}$. This implies for t' that $\bar{v}_{t'} \geq \varepsilon\beta^*$. Hence,

$$\sum_{t\in\mathcal{T}} x_t \bar{v}_t \geq \varepsilon x_{t'} \bar{v}_{t'} + \sum_{t\in\mathcal{T}} x_t \Big((1-\varepsilon)v_t - \alpha_t^* - \gamma^* s_t\Big) > \varepsilon\bar{v}_{t'} + \beta^* \geq (1+\varepsilon^2)\beta^*.$$

By definition of $\tilde{v}$, we have $\bar{v}_t - \tilde{v}_t \leq \varepsilon^4\beta^*$ and $\sum_{t\in\mathcal{T}} x_t(\bar{v}_t - \tilde{v}_t) \leq \varepsilon^3\beta^*$. This implies

$$\sum_{t\in\mathcal{T}} x_t \tilde{v}_t = \sum_{t\in\mathcal{T}} x_t \bar{v}_t - \sum_{t\in\mathcal{T}} x_t(\bar{v}_t - \tilde{v}_t) > (1+\varepsilon^2)\beta^* - \varepsilon^3\beta^* \geq \beta^*,$$

where the last inequality follows from $\varepsilon \leq 1$. By construction of the DP, x^* is an optimal solution for the instance induced by the values $\tilde{v}_t$ and multiplicities $\min\left\{\frac{1}{\varepsilon}, n_t\right\}$ and achieves a total value at most β^*. Therefore,

$$\beta^* \geq \sum_{t\in\mathcal{T}} x_t^* \tilde{v}_t \geq \sum_{t\in\mathcal{T}} x_t \tilde{v}_t > \beta^*,$$

which gives a contradiction. □

Proof of Lemma 7.14. As discussed above, the high-level idea is to solve (D), the dual of (P), with the Ellipsoid Method and to consider only the variables corresponding to constraints added by the Ellipsoid Method for solving (P).

As (S) is part of the separation problem for (D), there is no efficient way to exactly solve the separation problem, unless $\mathcal{P} = \mathcal{NP}$. Lemma 7.15 provides us a way to approximately solve the separation problem. As an approximately feasible solution for (D) cannot be directly used to determine the important variables in (P), we add an upper bound r on the objective function as a constraint to (D) and search for the largest r such that the Ellipsoid Method

returns infeasible. This implies that r is an *upper bound* on the objective function of (D) which in turn guarantees a *lower bound* on the objective function value of (P) by weak duality.

Of course, testing all possible values for r is intractable and we restrict the possible choices for r. Observe that $v_{\text{LP}} \in [v_{\max}, nv_{\max}]$ where v_{LP} is the optimal value of (P). Thus, for $k \in \mathbb{N}$ with $\lceil \log_{1+\varepsilon} v_{\max} \rceil \leq k \leq \lceil \log_{1+\varepsilon}(nv_{\max}) \rceil$, we use $r = (1+\varepsilon)^k$ as upper bound on the objective function. That is, we test if (D) extended by the objective function constraint

$$\lfloor (1-3\varepsilon)m \rfloor \beta + (\lfloor (1-3\varepsilon)m \rfloor S - s_S)\gamma + \sum_{t \in \mathcal{T}} n_t \alpha_t \leq r$$

is declared feasible by the Ellipsoid Method with the approximate separation oracle for (S). We refer to the feasibility problem by (D_r).

For a given solution $(\alpha^*, \beta^*, \gamma^*)$ of (D_r) the separation problem asks for one of the two: either the affirmation that the point is feasible or a separating hyperplane that separates the point from any feasible solution. The non-negativity of α_t^*, β^*, and γ^* an be checked in time $\mathcal{O}(|\mathcal{T}|) = \mathcal{O}\Big(\frac{\log^2 n}{\varepsilon^4}\Big)$. In case of a negative answer, the corresponding non-negativity constraint is a feasible separating hyperplane. Similarly, in time $\mathcal{O}(|\mathcal{T}|)$, we can check whether the objective function constraint $\lfloor (1-3\varepsilon)m \rfloor \beta + (\lfloor (1-3\varepsilon)m \rfloor S - s_S)\gamma + \sum_{t \in \mathcal{T}} n_t \alpha_t \leq r$ is violated and add it as a new inequality if necessary. In case the non-negativity and objective function constraints are not violated, the separation problem is given by the knapsack problem in (S). Algorithm 7.3 either outputs a configuration that yields a valid separating hyperplane or declares $(\alpha^*, \beta^*, \gamma^*)$ feasible, i.e., $\beta^* + s_c\gamma^* + \sum_{t \in \mathcal{T}} \alpha_t^* n_{tc} \geq (1-\varepsilon)v_c$ for all $c \in \mathcal{C}$. This implies that $(\alpha^*, \beta^*, \gamma^*)$ is feasible for the following LP. (Note that we changed the right side of the constraints when compared to (D).)

$$\begin{array}{lllll}
\min & \lfloor (1-3\varepsilon)m \rfloor \beta + (\lfloor (1-3\varepsilon)m \rfloor S - s_S)\gamma + \displaystyle\sum_{t \in \mathcal{T}} n_t \alpha_t & & & \\
\text{s.t.} & \beta + s_c\gamma + \displaystyle\sum_{t \in \mathcal{T}} \alpha_t n_{tc} & \geq & (1-\varepsilon)v_c & \text{for all } c \in \mathcal{C} \\
& \alpha_t & \geq & 0 & \text{for all } t \in \mathcal{T} \\
& \beta, \gamma & \geq & 0 &
\end{array} \qquad (\text{D}^{(1-\varepsilon)})$$

Let r^* be minimal such that (D_{r^*}) is declared feasible. Let $v_D^{(1-\varepsilon)}$ denote the optimal solution value of ($\text{D}^{(1-\varepsilon)}$). As $(\alpha^*, \beta^*, \gamma^*)$ is feasible with objective value at most r^*, we have $v_D^{(1-\varepsilon)} \leq r^*$. Let $v^{(1-\varepsilon)}$ denote the optimal solution value of its dual, i.e., of the following LP.

$$
\begin{array}{lllll}
\max & \displaystyle\sum_{c\in\mathcal{C}} y_c(1-\varepsilon)v_c & & & \\
\text{subject to} & \displaystyle\sum_{c\in\mathcal{C}} y_c s_c & \leq & \lfloor(1-3\varepsilon)m\rfloor S - s_S & \\
& \displaystyle\sum_{c\in\mathcal{C}} y_c & \leq & \lfloor(1-3\varepsilon)m\rfloor & \qquad (\mathrm{P}^{(1-\varepsilon)}) \\
& \displaystyle\sum_{c\in\mathcal{C}} y_c n_{tc} & \leq & n_t & \text{for all } t\in\mathcal{T} \\
& y_c & \geq & 0 & \text{for all } c\in\mathcal{C}
\end{array}
$$

Then, $y = 0$ is feasible for $(\mathrm{P}^{(1-\varepsilon)})$, and by weak duality, we have

$$v^{(1-\varepsilon)} \leq v_D^{(1-\varepsilon)} \leq r^*.$$

Note that (P) and $(\mathrm{P}^{(1-\varepsilon)})$ have the same feasible region and their objective functions only differ by a factor $(1-\varepsilon)$. This implies that

$$v_{\mathrm{LP}} = \frac{v^{(1-\varepsilon)}}{1-\varepsilon} \leq \frac{r^*}{1-\varepsilon}. \tag{7.1}$$

Because of this relation between v_{LP} and r^* it suffices to find a feasible solution for (P) with objective function value close to r^* in order to prove the lemma.

To this end, let $\mathcal{C}_r$ be the configurations that correspond to the inequalities added by the Ellipsoid Method while solving (D_r) for $r = \frac{r^*}{1+\varepsilon}$. Consider the problems (P) and (D) restricted to the variables y_c, for $c \in \mathcal{C}_r$, and to the constraints corresponding to $c \in \mathcal{C}_r$, respectively, and denote these restricted LPs by (P′) and (D′). Let v' and v'_D be their respective optimal values.

It holds that $v'_D > r$ as the Ellipsoid Method also returns infeasibility for (D′) when run on (D′) extended by the objective function constraint for r. As $y = 0$ is feasible for (P′) and $\alpha = 0$, $\beta = \max_{c\in\mathcal{C}_r} v_c$, and $\gamma = 0$ are feasible for (D′), their objective function values coincide by strong duality, i.e., $v' = v'_D > r$. If we have an optimal solution to (P′), then this solution is also feasible for (P) and achieves an objective function value

$$v' > \frac{r^*}{1+\varepsilon} \geq \frac{1-\varepsilon}{1+\varepsilon} v_{\mathrm{LP}},$$

where we used Equation (7.1) for the last inequality.

It remains to show that the Ellipsoid Method can be applied to the setting presented here and that the running time of the just described algorithm is indeed bounded by a polynomial in $\log n$, $\frac{1}{\varepsilon}$, and $\log U$. Recall that U is an upper bound on the absolute values of the denominators and numerators appearing in (D), i.e., on Sm and $nv_{\max}$. Observe that by Lemma 7.15, the separation oracle runs in time $\mathcal{O}\Big(\frac{\log^4 n}{\varepsilon^{14}}\Big)$. The number of iterations of the Ellipsoid Method will be bounded by a polynomial in $\log U$ and $\tilde{n} \in \mathcal{O}\Big(\frac{\log^2 n}{\varepsilon^4}\Big)$. Here, $\tilde{n}$ is an upper bound on the number of variables in the problems (D_r) (and hence also $(\mathrm{D}^{(1-\varepsilon)})$).

The feasible region of (D_r) is a subset of the feasible region of $(D^{(1-\varepsilon)})$, even when the objective function constraint is added to the latter LP. The Ellipsoid Method usually is applied to full-dimensional, bounded polytopes that guarantee two bounds: If the polytope is non-empty, then its volume is at least $v > 0$. The polytope is contained in a ball of volume at most V. As shown in the book by Bertsimas and Tsitsiklis [BT97], these assumptions can always be ensured and the parameters v and V can be chosen as polynomial functions of $\tilde{n}$ and U. Since we cannot check feasibility of (D_r) directly, we choose the parameters v and V as described in [BT97, Chapter 8] for the problem $(D^{(1-\varepsilon)})$ extended by the objective function constraint for r. After $N = \mathcal{O}\left(\tilde{n} \log \frac{V}{v}\right)$ iterations, the modified Ellipsoid Method either finds a feasible solution to $(D^{(1-\varepsilon)})$ with objective function value at most r or correctly declares (D_r) infeasible. In [BT97, Chapter 8] it is shown that the number of iterations N satisfies $N = \mathcal{O}(\tilde{n}^4 \log(\tilde{n}U))$ and that the overall running time is polynomially bounded in $\tilde{n}$ and $\log U$.

Hence, (P'), the problem (P) restricted to variables corresponding to constraints added by the Ellipsoid Method, has at most N variables and, thus, a polynomial time algorithm for linear programs can be applied to (P') to obtain an optimal solution in time $\left(\frac{\log U}{\varepsilon}\right)^{\mathcal{O}(1)}$. □

Obtaining an integral solution Next, we show how to turn a fractional solution to a particular class of packing LPs into a feasible solution using some additional knapsacks given by resource augmentation. The LP relaxation of the configuration ILP considered here belongs to this class of LPs, and the assumption $m \geq \frac{16}{\varepsilon^7} \log^2 n$ ensures that we can round a basic feasible solution to an integral packing of big items using at most $\lfloor (1-2\varepsilon)m \rfloor$ knapsacks.

Formally, we consider a packing problem of items into a given set K of knapsacks with capacities S_i. These knapsacks are grouped to obtain the set $\mathcal{G}$ where group $g \in \mathcal{G}$ contains m_g knapsacks and has total capacity S_g. The objective is to maximize the total value without violating any capacity constraint. Each item j has a certain type t, i.e., value $v_j = v_t$ and size $s_j = s_t$, and in total there are n_t items of type t. Items can either be packed as single items or as part of configurations. A configuration c, that packs $n_{c,t}$ items of type t, has total value $v_c = \sum_t n_{c,t} v_t$ and size $s_c = \sum_t n_{c,t} s_t$. The set E represents the items and the configurations that we are allowed to pack for maximizing the total value. Without loss of generality, we assume that for each element $e \in E$ there exists at least one knapsack i where this element fits, i.e, $s_e \leq S_i$.

Let $0 \leq \delta \leq 1$ and $s \geq 0$. Later we will choose $\delta = 1 - \Theta(\varepsilon)$ since intuitively an $\Theta(\varepsilon)$-fraction of the knapsacks remains unused. Consider the packing ILP for the above described problem with variables $z_{e,g}$, where $e \in E$ and $g \in \mathcal{G}$. The ILP may additionally contain constraints of the form

$$\sum_{e \in E, g \in \mathcal{G}'} s_e z_{e,g} \leq \delta \sum_{g \in \mathcal{G}'} S_g - s \text{ and } \sum_{e \in E', g \in \mathcal{G}'} z_{e,g} \leq \delta \sum_{g \in \mathcal{G}'} m_g \,,$$

i.e., the elements assigned to a subset of knapsack types $\mathcal{G}'$ do not violate the total capacity of

a δ-fraction of the knapsacks in $\mathcal{G}'$ while reserving a space of size s and a particular subset E' of these elements uses at most a δ-fraction of the available knapsacks.

Let $v(z)$ be the value attained by a certain solution z and let $n(z)$ be the number of non-zero variables of z. The following lemma shows that there is an integral solution of value at least $v(z)$ using at most $n(z)$ extra knapsacks. The high-level idea of the proof is to round down each non-zero variable $z_{e,g}$ and pack the corresponding elements as described by $z_{e,g}$. For achieving enough value, we additionally place one extra element e into the knapsacks given by resource augmentation for each variable $z_{e,g}$ that was subjected to rounding.

More precisely, for each element e and each knapsack group g, we define $\bar{z}'_{e,g} = \lfloor z_{e,g} \rfloor$ and $\bar{z}''_{e,g} = \lceil z_{e,g} - \bar{z}'_{e,g} \rceil$. Note that $\bar{z}' + \bar{z}''$ may require more items of a certain type than are available. Hence, for each item type t that is now packed more than n_t times, we reduce the number of items of type t in $\bar{z}' + \bar{z}''$ by either adapting the chosen configurations if t is packed in a configuration or by decreasing the variables of type $z_{t,g}$ if items of type t are packed as single items in knapsacks of group g. Let z' and z'' denote the solutions obtained by this transformation. For some elements e, the packing described by $z'_{e,g} + z''_{e,g}$ may now use more or less elements than $z_{e,g}$ due to the just described reduction of items.

Lemma 7.16. *Any fractional solution z to the packing ILP described above can be rounded to an integral solution with value at least $v(z)$ using at most $n(z)$ additional knapsacks of capacity $\max_{i \in K} S_i$.*

Proof. Consider a particular item type t. If $\bar{z}' + \bar{z}''$ packs at most n_t items of this type, then the value achieved by z for this particular item type is upper bounded by the value achieved by $z' + z''$. If an item type was subjected to the modification, then $z' + z''$ packs exactly n_t items of this type while z packs at most n_t items. This implies that $v(z' + z'') \geq v(z)$.

It remains to show how to pack $\bar{z}' + \bar{z}''$ (and, thus, $z' + z''$) into the knapsacks given by K and potentially $n(z)$ additional knapsack. Clearly, $\bar{z}'$ can be packed exactly as z was packed. If $z_{e,g} = 0$ for $e \in E$ and $g \in \mathcal{G}$, then $\bar{z}'_{e,g} = 0$. Hence, the number of non-zero entries in $\bar{z}''$ is bounded by $n(z)$. Consider one element $e \in E$ and a knapsack group g with $\bar{z}''_{e,g} = 1$ and let i be a knapsack where e fits. Pack e into i.

Since reducing the number of packed items of a certain type only decreases the size of the corresponding configuration or the number of individually packed elements, the solution $z' + z''$ can be packed exactly as described for $\bar{z}' + \bar{z}''$. Therefore, we need at most $n(z)$ extra knapsacks to pack z'', which concludes the proof. □

Having found a feasible solution with the Ellipsoid Method, we use Gaussian elimination to obtain a basic feasible solution with no worse objective function value. We note that this procedure has a running time bounded by $(N|\mathcal{T}|)^{\mathcal{O}(1)}$, where N is the number of non-zero variables in the solution found by the Ellipsoid Method. Since basic feasible solutions have at most $|\mathcal{T}| + 2$ non-vanishing variables, the assumptions $\frac{16}{\varepsilon^7} \log^2 n \leq m$ and $m < n$

imply $\frac{16}{\varepsilon^7}\log^2 m \leq m$. This in turn guarantees $|\mathcal{T}| + 2 \leq \lfloor \varepsilon m \rfloor$. Hence, rounding the solution as described above uses at most $\lfloor (1-2\varepsilon)m \rfloor$ knapsacks and achieves a value of at least v_{LP}.

Corollary 7.17. *If $\frac{16}{\varepsilon^7}\log^2 n \leq m$, any feasible solution of the LP relaxation of* (P) *with at most N non-zero variables can be rounded to an integral solution using at most $\lfloor (1-2\varepsilon)m \rfloor$ knapsacks with total value at least v_{LP} in time $(N|\mathcal{T}|)^{\mathcal{O}(1)}$.*

Given an integral packing of big items, we explain how to pack small items, i.e., items with $s_j < \varepsilon S$, using resource augmentation. More precisely, let K be a set of knapsacks and let $\mathcal{J}'_S \subseteq \mathcal{J}$ be a subset of items that are small with respect to every knapsack in K. Let $\mathcal{J}' \subset \mathcal{J}$ be a set of items admitting an integral packing into $m = |K|$ knapsacks that preserves a space of at least $s(\mathcal{J}'_S)$ in these m knapsacks. We develop a procedure to extend this packing to an integral packing of all items $\mathcal{J}' \cup \mathcal{J}'_S$ in $\lceil (1+\varepsilon)m \rceil$ knapsacks where the $\lceil \varepsilon m \rceil$ additional knapsacks can be chosen to have the smallest capacity of knapsacks in K.

We use a packing approach similar to Next Fit for the problem Bin Packing. That is, consider an arbitrary order of the small items and an arbitrary order of the knapsacks filled with big items. We open the first knapsack in this order for small items. If the next small item j still fits into the open knapsack, we place it there and decrease the remaining capacity accordingly. If it does not fit anymore, we pack this item into the next empty slot of an additional knapsacks (possibly opening a new one), close the current original knapsack, and open the next one for packing small items. We call such an item *cut*.

Lemma 7.18. *The procedure described above feasibly packs all items $\mathcal{J}' \cup \mathcal{J}'_S$ in $\lceil (1+\varepsilon)m \rceil$ knapsacks where the $\lceil \varepsilon m \rceil$ additional knapsacks can be chosen to have the smallest capacity of knapsacks in K.*

Proof. We start by showing that all small items are packed after the last original knapsack is closed. Toward a contradiction, suppose that there is a small item j left *after* all original knapsacks were closed while packing small items. As a knapsack is only closed if the current small item does not fit anymore, this implies that the volume of all small items that are packed so far have a total volume at least as large as the total remaining capacity of knapsacks in K after packing $\mathcal{J}'$. Since j is left unpacked after all original knapsacks have been closed, the total volume of all items in $\mathcal{J}' \cup \mathcal{J}'_S$ is strictly larger than the total capacity of the original knapsacks in K. This contradicts the assumption imposed on $\mathcal{J}'_B$ and on $\mathcal{J}'_S$. Hence, all items in $\mathcal{J}'_S$ are packed. Therefore, the packing created by the procedure is integral and feasible.

It remains to bound the number of additional knapsacks. Observe that each item that we packed into a knapsack given by resource augmentation while an original knapsack was still available, implied the closing of the current knapsack and the opening of a new one. Hence, for each original knapsack at most one small item was placed into the additional knapsacks. Thus, at most m small items are packed into the additional knapsacks. Since by definition of small items at least $\frac{1}{\varepsilon}$ items fit into one additional knapsack, we only need $\lceil \varepsilon m \rceil$ extra knapsacks for such items. □

Bounding the performance and the running time

Lemma 7.19. *Let P_F be the solution returned by Algorithm 7.2 and let* Opt *be a current optimal solution. It holds that $v(P_F) \geq \frac{(1-\varepsilon)^2(1-2\varepsilon)(1-4\varepsilon)}{(1+\varepsilon)^4} v(\text{Opt})$.*

Proof. Fix Opt. The solution found by our algorithm achieves the maximal value over all combinations of guesses v_S, the value contributed by small items, and of $\ell_{\max}$, the largest index of a value class of a big item in Opt. Thus, it suffices to find a combination of v_S and $\ell_{\max}$ such that P, the corresponding packing, is feasible and satisfies $v(P) \geq \frac{(1-\varepsilon)^2(1-2\varepsilon)(1-4\varepsilon)}{(1+\varepsilon)^4} v(\text{Opt})$.

Let Opt_B be the set of big items in Opt, let $\ell_{\max} := \max\{\ell : V_\ell \cap \text{Opt}_B \neq \emptyset\}$, and let $\text{Opt}_{\mathcal{T}}$ be the most valuable packing after linear grouping with $\ell_{\max}$. For this guess $\ell_{\max}$, let $P_S \cup \{j^\star\}$ be the set of small items of Lemma 7.13 such that $v_{\text{ILP}} + v(P_S) + v_{j^\star} \geq \frac{1-4\varepsilon}{1+\varepsilon} v(\text{Opt}_{\mathcal{T}})$. By Corollary 7.17, there is a set of big items P_B with a feasible packing into $\lfloor (1-2\varepsilon)m \rfloor$ knapsacks with total value at least $\frac{1-\varepsilon}{1+\varepsilon} v_{\text{ILP}}$. Packing $j^\star$ on its own and P_S following a Next-Fit-like algorithm, we extend this to a feasible packing of $P_B \cup P_S \cup \{j^\star\}$ into $\lceil (1+\varepsilon)\lfloor (1-2\varepsilon)m \rfloor \rceil + 1$ knapsacks; see Lemma 7.18. Due to the assumption $m \geq \frac{16}{\varepsilon^7} \log^2 n$, we can bound the number of total knapsacks indeed by m. With Lemma 7.13,

$$v(P_F) \geq v(P) \geq \frac{1-\varepsilon}{1+\varepsilon} v_{\text{ILP}} + v_S + v_{j^\star} \geq \frac{(1-\varepsilon)(1-4\varepsilon)}{(1+\varepsilon)^2} v(\text{Opt}_{\mathcal{T}}).$$

With Corollary 7.12 we get

$$v(P_F) \geq \frac{(1-\varepsilon)^2(1-2\varepsilon)(1-4\varepsilon)}{(1+\varepsilon)^4} v(\text{Opt}),$$

which concludes the proof. □

The next lemma bounds the running time of our algorithm. The proof follows from the fact that the algorithm considers at most $\mathcal{O}(\log_{1+\varepsilon} v_{\max})$ guesses for $\ell_{\max}$ and $\mathcal{O}(\log_{1+\varepsilon} n v_{\max})$ guesses for v_S, the running time for dynamic linear grouping bounded in Lemma 7.10, and the running time for solving the configuration ILP as described in Lemma 7.14 and Corollary 7.17.

Lemma 7.20. *Let $U := \max\{Sm, nv_{\max}\}$. The running time of our algorithm is bounded by $\left(\frac{\log U}{\varepsilon}\right)^{\mathcal{O}(1)}$.*

Answering Queries Note that, throughout the course of the dynamic algorithm, we only implicitly store solutions. In the remainder of this section, we explain how to answer the queries stated in Section 7.2 and bound the running times of the corresponding algorithms. We refer to the time frame between two updates as a *round* and introduce a counter τ that is increased after each update and denotes the current round. Since answers to queries have to stay consistent in a round, we *cache* existing query answers by additionally storing a round $t(j)$ and a knapsack $k(j)$ for each item in the data structure for items where $t(j)$ stores the last

round in which item j has been queried and $k(j)$ points to the knapsack of j in round $t(j)$. Storing $t(j)$ is necessary since resetting the cached query answers after each update takes too much running time. If j was not selected in $t(j)$, we store and return this with $k(j) = 0$.

Let $\bar{y}_c$, for $c \in \mathcal{C}$, be the packing for the big items in terms of the variables of the configuration ILP. During the Ellipsoid Method and the rounding of the fractional solution to an integral solution, the set $\overline{\mathcal{C}} := \{c \in \mathcal{C} : \bar{y}_c \geq 1\}$ was constructed. We assume that this set is ordered in some way and stored in a list. In the following we use the position of $c \in \overline{\mathcal{C}}$ in that list as the index of c. For assigning $\bar{y}_c$ distinct knapsacks to $c \in \overline{\mathcal{C}}$ we use the ordering of the configurations and map the knapsacks $\sum_{c'=1}^{c-1} \bar{y}_{c'} + 1, \ldots, \sum_{c'=1}^{c} \bar{y}_{c'}$ to c.

For small items, we store all items in a balanced binary search tree sorted by non-increasing density. For simplicity, let $P_S = \{1, \ldots, j^\star - 1\}$ be the set of items (sorted by non-increasing density) that translate the guess v_S into the size s_S of small items in the current solution. Item $j^\star$ is packed into its own knapsack. Any item $j \leq j^\star - 1$ is either packed regularly into the empty space of a knapsack with a configuration or it is packed into a knapsack designated for packing cut small items. Therefore, we maintain two pointers: κ^r points to the next knapsack where a small item is supposed to go if it is packed *regularly* and κ^c points to the knapsack where the next *cut* small item is packed. We initialize these values with $\kappa^r = 1$ and $\kappa^c = \lfloor (1 - 2\varepsilon)m \rfloor + 1$. To determine if an item is packed regularly or as cut item, we store in ρ^r the remaining capacity of κ^r initialized with $\kappa^r = S - s_1$ where s_1 is the size of the first configuration in $\overline{\mathcal{C}}$. We store in ρ^c the remaining slots of small items in knapsack κ^c and initialize this with $\rho^c = \frac{1}{\varepsilon}$.

For each type t of big items, we maintain a pointer κ_t to the knapsack where the next queried item of type t is supposed to be packed. Moreover, the counter η_t stores how many slots κ_t still has available for items of type t. These two values are initialized with the first knapsack that packs items of type t and $\eta_t = n_{c,t}$ where c is the configuration of κ_t. If no items of type t are packed, we set $\kappa_t = 0$. Let $\bar{n}_t$ denote the number of items of type t belonging to solution $\bar{y}$. We will only pack the first, i.e., smallest, $\bar{n}_t$ items of type t. Figure 7.4 depicts the pointers and counters after some items already have been queried.

Consider a queried small item j. If $t(j) = \tau$, we return $k(j)$. Otherwise, set $t(j) = \tau$ and determine whether j is currently part of the solution. If j does not belong to the densest $j^\star$ items, we return $k(j) = 0$. Otherwise, we determine where j is packed. If $j = j^\star$, we return $k(j) = m$. Else, we figure out whether j is packed into the knapsack κ^r or into κ^c. If $\rho^r \geq s_j$, we simply update ρ^r to $\rho^r - s_j$ and return $k(j) = \kappa^r$. Otherwise, we decrease ρ^c by one and pack j as cut item in κ^c. If $\rho^c = 0$ holds after the update, we increase κ^c by one and set $\rho^c = \frac{1}{\varepsilon}$. Further, we need to close κ^r and update κ^r and ρ^r accordingly. To this end, we increase κ^r by one and determine ρ^r, the remaining capacity in knapsack κ^r. Then, we return $k(j)$.

Consider a queried big item j. If $t(j) = \tau$, we return $k(j)$. Otherwise, we set $t(j) = \tau$ and compute whether item j is packed by the current solution. Let V_ℓ be the value class of j. If $\ell \notin \{\ell_{\min}, \ldots, \ell_{\max}\}$, we return $k(j) = 0$. Otherwise, we retrieve the type t of item j.

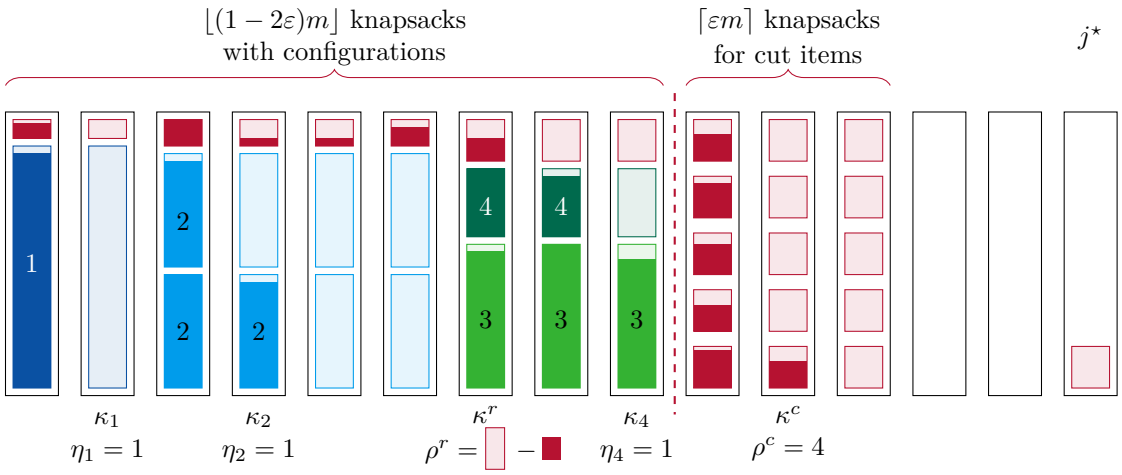

Figure 7.4: Pointers and counters used for answering queries: Lightly colored rectangles represent slots to be filled with items. Big (blue and green) items are packed one item per slot. Item type 3 does not have any slots left. Small (red) items are packed either until the slot is filled (left side) or one item per slot (right side). The not yet queried, small item $j^\star$ gets its own knapsack.

Given t, we determine if j belongs to the first $\bar{n}_t$ items of type t. If this is not the case, we return $k(j) = 0$. If this is the case, then we set $k(j) = \kappa_t$ instead and we decrease η_t by one. If this remains non-zero, we return $k(j) = \kappa_t$. Otherwise, we find the next knapsack that packs items of type t and update κ_t and η_t accordingly before returning $k(j)$. We summarize this algorithm in Algorithm 7.4.

Algorithm 7.4: Answering item queries in round τ

```
if t(j) ≠ τ do
    t(j) ← τ
    if s_j < εS do // small item
        if  j > j* do
            k(j) ← 0 // not selected
        else-if j = j* do
            k(j) ← m
        else-if s_j ≤ ρ^r do
            k(j) ← κ^r
            ρ^r ← ρ^r − s_j
        else
            k(j) ← κ^c
            κ^r ← κ^r + 1; update ρ^r
            ρ^c ← ρ^c − 1
            update κ^c and ρ^c if necessary
    else // big item
        ℓ ← value class of j
```

```
        if ℓ ∉ {ℓ_min, ..., ℓ_max} do
            k(j) ← 0 // not selected
        else
            t ← item type of j
            if j not among the first n̄_t items of type t do
                k(j) ← 0 // not selected
            else
                k(j) = κ_t
                η_t ← η_t − 1
                update κ_t and η_t if necessary
return k(j)
```

For being able to return the solution value in constant query time, we actually compute the solution value once after each update operation and store it. More precisely, the value achieved by the small items, v_S can be computed with a prefix computation of the first $j^\star$ items in the density-sorted tree for small items. For computing the value of big items, we consider each value class V_ℓ with $\ell \in \{\ell_{\min}, \dots, \ell_{\max}\}$ individually. Per value class and per item type, we use prefix computation to determine the value v_t of the first $\bar{n}_t$ items of type t. Lemma 7.25 guarantees that the running time is indeed upper bounded by the update time and, thus, does not change the order of magnitude described in Lemma 7.20.

When queried the complete solution, we return a list of packed items together with their respective knapsacks. To this end, we start by querying the $j^\star$ densest small items using the algorithm for item queries. For big items, we query the first $\bar{n}_t$ items of each item type $t \in \mathcal{T}$.

We prove the parts of the following lemmas individually.

Lemma 7.21. *The solution determined by the query algorithms is feasible and achieves the claimed total value. The query times of our algorithm are as follows:*

(i) Single item queries can be answered in time $\mathcal{O}\Big(\log n + \max\Big\{\log \frac{\log n}{\varepsilon}, \frac{1}{\varepsilon}\Big\}\Big)$

(ii) solution value queries can be answered in time $\mathcal{O}(1)$*, and*

(iii) queries of the entire solution P *can be answered in time* $\mathcal{O}\Big(|P|\frac{\log^4 n}{\varepsilon^4}\log\frac{\log n}{\varepsilon}\Big)$.

Lemma 7.22. *The solution determined by the query algorithms is feasible and achieves the claimed total value.*

Proof. By construction of $t(j)$ and $k(j)$, the answers to queries happening between two consecutive updates are consistent.

For small items, observe that $1, \dots, j^\star$ are the densest small items in the current instance. By Lemma 7.18, the packing obtained by our algorithms is feasible for these items. In Lemma 7.19 we argue that these items contribute enough value to our solution.

For big items, we observe that their actual size is at most the size of their item types. Hence, packing an item of type t where the implicit solution packs an item of type t is feasible. The

algorithms correctly pack the first $\bar{n}_t$ items of type t. A knapsack with configuration $c \in \overline{\mathcal{C}}$ correctly obtains $n_{c,t}$ items of type t. Moreover, each configuration $c \in \overline{\mathcal{C}}$ gets assigned $\bar{y}_c$ knapsacks. Hence, the algorithm packs exactly the number of big items as dictated by the implicit solution $\bar{y}$. □

Lemma 7.23. *The data structures for big items can be generated in time* $\mathcal{O}\left(\frac{\log^4 n}{\varepsilon^9}\right)$. *Queries for big items can be answered in time* $\mathcal{O}\left(\log n + \log \frac{\log n}{\varepsilon}\right)$.

Proof. We assume that $\overline{\mathcal{C}}$ is already stored in some list. We start by formally mapping knapsacks to configurations. To this end, we create a list $\alpha = (\alpha_c)_{c \in \overline{\mathcal{C}}}$, where $\alpha_c = \sum_{c'=1}^{c-1} \bar{y}_{c'}$ is the first knapsack with configuration $c \in \overline{\mathcal{C}}$. Using $\alpha_c = \alpha_{c-1} + \bar{y}_{c-1}$, we can compute these values in constant time. Hence, by iterating once through $\overline{\mathcal{C}}$, list α can be generated in $\mathcal{O}(|\overline{\mathcal{C}}|)$.

We start by recomputing the indices needed for the dynamic linear grouping approach. For each value class V_ℓ with $\ell \in \{\ell_{\min}, \dots, \ell_{\max}\}$, we access the items corresponding to the boundaries of the item types $\mathcal{T}_\ell$ in order to obtain the item types $\mathcal{T}_\ell$. By construction, these types are already ordered by non-decreasing size s_t. By Lemma 7.10, these item types can be computed in time $\mathcal{O}\left(\frac{\log^4 n}{\varepsilon^4}\right)$ and stored in one list $\mathcal{T}_\ell$ per value class V_ℓ.

For maintaining and updating the pointer κ_t, we generate a list $\mathcal{C}_t$ of all configurations $c \in \overline{\mathcal{C}}$ with $n_{c,t} \geq 1$. By iterating through each $c \in \overline{\mathcal{C}}$, we can add c to the list of t if $n_{c,t} \geq 1$. We additionally store $n_{c,t}$ and α_c in the list $\mathcal{C}_t$. While iterating through the configurations, we additionally compute $\bar{n}_t = \sum_{c \in \overline{\mathcal{C}}} \bar{y}_c n_{c,t}$ and store $\bar{n}_t$ in the same list as the item types $\mathcal{T}_\ell$. Note that, since the list of $\overline{\mathcal{C}}$ is ordered by index, the created lists $\mathcal{C}_t$ are also sorted by index. For each item type, we point κ_t to the first knapsack of the first added configuration c and set $\eta_t = n_{c,t}$. If the list of an item type remains empty, we set $\kappa_t = 0$. Since each configuration contains at most $\frac{1}{\varepsilon}$ item types, the lists $\mathcal{C}_t$ can be generated in time $\mathcal{O}\left(\frac{|\overline{\mathcal{C}}||\mathcal{T}|}{\varepsilon}\right)$.

Now consider a queried big item j. In time $\mathcal{O}(\log n)$, we can decide whether j has already been queried in the current round. If not, let V_ℓ be the value class of j, which was computed upon arrival of j. If $\ell \notin \{\ell_{\min}, \dots, \ell_{\max}\}$, then j does not belong to the current solution and no data structures need to be updated. Otherwise, the type of j is determined by accessing the item types $\mathcal{T}_\ell$ in time $\mathcal{O}\left(\log \frac{\log n}{\varepsilon}\right)$. Once t is determined, $\bar{n}_t$ can be added to the left boundary of type t in order to determine if j is packed or not. If j belongs to the current solution, pointer κ_t dictates the answer to the query.

In order to update κ_t and η_t, we extract c, the configuration of knapsack κ_t in time $\mathcal{O}(\log |\overline{\mathcal{C}}|)$ by binary search over the list α. If $\kappa_t + 1 < \alpha_{c+1}$, then κ_t is increased by one and η_t set to $n_{c,t}$ in constant time. If not, then the next configuration c' containing t can be found with binary search over the list $\mathcal{C}_t$ in time $\mathcal{O}(\log |\overline{\mathcal{C}}|)$. If no such configuration is found, we set $\kappa_t = 0$. Otherwise, we set $\kappa_t = \alpha_{c'}$ and $\eta_t = n_{c',t}$. Overall, queries for big items can be answered in time $\mathcal{O}\left(\max\left\{\log |\overline{\mathcal{C}}|, \log \frac{\log n}{\varepsilon}\right\}\right)$.

Observing that $|\overline{\mathcal{C}}| \in \mathcal{O}(|\mathcal{T}|) = \mathcal{O}\left(\frac{\log^2 n}{\varepsilon^4}\right)$ completes the proof. □

Lemma 7.24. *Given the data structures for big items, the data structures for small items can be generated in time* $\mathcal{O}\left(\log \frac{\log n}{\varepsilon}\right)$. *The running time for answering queries for small items is* $\mathcal{O}\left(\log n + \max\left\{\log \frac{\log n}{\varepsilon}, \frac{1}{\varepsilon}\right\}\right)$.

Proof. We initialize $\kappa^r = 1$ and $\rho = S - s_1$ where s_1 is the total size of the configuration assigned to the first knapsack. For packing cut items, we use the pointer κ^c to the current knapsack for cut items while ρ^c stores the remaining slots of small items. We initialize these values with $\kappa^c = \lfloor(1-2\varepsilon)m\rfloor + 1$ and $\rho^c = \frac{1}{\varepsilon}$. These initializations can be computed in time $\mathcal{O}(\log|\overline{\mathcal{C}}|)$ (for extracting s_1).

Now consider a queried small item j. In time $\mathcal{O}(\log n)$ we can decide whether j has already been queried in the current round. In constant time, we can decide whether $j > j^\star$. If $j > j^\star$, the answer is NOT SELECTED. If $j = j^\star$, we return m. If $j < j^\star$, the algorithm only needs to decide if j is packed into κ^r or κ^c, which can be done in constant time. Finally, κ^r and κ^c as well as ρ^r and ρ^c need to be updated. While κ^c, κ^r, and ρ^c can be updated in constant time, we need to compute the configuration c and remaining capacity $S - s_c$ of knapsack κ^r when the pointer is increased. By using binary search over the list α, the configuration can be determined in time $\mathcal{O}(\log|\overline{\mathcal{C}}|)$. Once the configuration is known, ρ^r can be calculated in time $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$. Overall, queries for small items can be answered in time $\mathcal{O}\left(\log n + \max\left\{\log|\overline{\mathcal{C}}|, \frac{1}{\varepsilon}\right\}\right)$.

Using that $|\overline{\mathcal{C}}| \in \mathcal{O}(|\mathcal{T}|) = \mathcal{O}\left(\frac{\log^2 n}{\varepsilon^4}\right)$ concludes the proof. □

Lemma 7.25. *The total solution value can be computed in* $\mathcal{O}\left(\frac{\log^3 n}{\varepsilon^4}\right)$. *A query for the solution value can be answered in time* $\mathcal{O}(1)$.

Proof. The true value $\tilde{v}_S$ achieved by the small items can be determined by computing the prefix of the first $j^\star$ items in the density-sorted tree for small items in time $\mathcal{O}(\log n)$ by Lemma 7.2.

For computing the value of a big item, we consider each value class V_ℓ with $\ell \in \{\ell_{\min}, \dots, \ell_{\max}\}$ individually. There are at most $\mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right)$ many value classes by Lemma 7.6. For one value class, in time $\mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right)$, iterate through the item types t. For each item type, we can access the total value of the first $\bar{n}_t$ items in time $\mathcal{O}(\log n)$ by Lemma 7.2.

As these running times are subsumed by the running time of the update operation, we actually compute the solution value once after each update operation and store the value allowing for constant running time to answer the query. □

Lemma 7.26. *A query for the complete solution can be answered in time* $\mathcal{O}\left(|P|\frac{\log^4 n}{\varepsilon^4}\log\frac{\log n}{\varepsilon}\right)$, *where* P *is the set of items in our solution.*

Proof. The small items belonging to P can be accessed in time $\mathcal{O}(j^\star \log n)$ by Lemma 7.2. By Lemma 7.24, their knapsacks can be determined in time $\mathcal{O}\left(\log n + \max\left\{\log\frac{\log n}{\varepsilon}, \frac{1}{\varepsilon}\right\}\right)$.

For big items, we consider again at most $\mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right)$ many value classes individually. In time $\mathcal{O}\left(\frac{\log n}{\varepsilon^2}\right)$, we access the boundaries of the corresponding item types. In time $\mathcal{O}(\bar{n}_t \log n)$, we can access the $\bar{n}_t$ items of type t belonging to our solutions by Lemma 7.2. Lemma 7.23 ensures that their knapsacks can be determined in time $\mathcal{O}\left(\log n + \log \frac{\log n}{\varepsilon}\right)$.

In total, this bounds the running time by $\mathcal{O}\left(|P| \frac{\log^4 n}{\varepsilon^4} \log \frac{\log n}{\varepsilon}\right)$. □

Proof of main result

Proof of Theorem 7.11. In Lemma 7.19, we calculate the approximation ratio achieved by our algorithm. Lemma 7.20 gives the desired bounds on the update time while Lemma 7.21 bounds the time needed for answering a query. Lemma 7.21 also guarantees that the query answers are correct and consistent between two updates. □

7.5 Ordinary Knapsacks When Solving Multiple Knapsack

In this section, we consider instances for Multiple Knapsack with many knapsacks and arbitrary capacities. We show how to efficiently maintain a $(1+\varepsilon)$-approximation when given, as resource augmentation, L additional knapsacks that have the same capacity as a largest knapsack in the input instance, where $L \in \left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}$. While we may pack items into the additional knapsacks, an optimal solution is not allowed to use them. The algorithm will again solve the LP relaxation of a configuration ILP and round the obtained solution to an integral packing. However, in contrast to the problem for identical knapsacks, not every configuration fits into every knapsack and we therefore cannot just reserve a fraction of knapsacks in order to pack the rounded configurations since the knapsack capacities might not suffice. For this reason, we employ resource augmentation in the case of arbitrary knapsack capacities.

Again, we assume that item values are rounded to powers of $(1+\varepsilon)$ which results in value classes V_ℓ of items with value $v_j = (1+\varepsilon)^\ell$. We prove the following theorem.

Theorem 7.27. *For every $\varepsilon > 0$, there is a dynamic algorithm for* Multiple Knapsack *that, when given L additional knapsacks as resource augmentation, achieves an approximation factor of $(1+\varepsilon)$ with update time $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1/\varepsilon)} (\log m \log S_{\max} \log v_{\max})^{\mathcal{O}(1)}$. Item queries are answered in time $\mathcal{O}\left(\log m + \frac{\log n}{\varepsilon^2}\right)$, and the solution P that is maintained by our algorithm can be output in time $\mathcal{O}\left(|P| \frac{\log^3 n}{\varepsilon^4}\left(\log m + \frac{\log n}{\varepsilon^2}\right)\right)$.*

7.5.1 Algorithm

Data structures In this section, we maintain three different types of data structures. For storing every item j together with its size s_j, its value v_j, and the index of its value class ℓ_j, we maintain one balanced binary search tree where the items are sorted by non-decreasing time of arrival. For each value class V_ℓ, we maintain one balanced binary tree for sorting the items

with $\ell_j = \ell$ in order of non-decreasing size. We store the knapsacks sorted in non-increasing capacity in one balanced binary tree.

Algorithm The algorithm we develop in this section is quite similar to the dynamic algorithm for Multiple Knapsack with identical capacities. First, we use dynamic linear grouping for the current set of items to obtain item types. However, in contrast to identical knapsacks, one particular item may be big with respect to one knapsack, small with respect to another, and may not even fit in a third knapsack. Thus, we use the item types to partition the knapsacks into groups to simulate knapsacks with identical capacities. Within one group, we give an explicit packing of the big items into slightly less knapsacks than belonging to the group by solving a configuration ILP. For packing small items, we would like to use a guess of the size of small items per groups and later use again Next Fit to pack them integrally. However, since items classify as big in one knapsack group and as small in another group, instead of guessing the size of small items per knapsack group, we incorporate their packing into the configuration ILP by reserving sufficient space for the small items in each group. More precisely, we assign items as big items via configurations or as small items by number to the various groups. The remainder of the algorithm is straight-forward: we relax the integrality constraint to find a fractional solution and use the tools developed in Lemmas 7.16 and 7.18 to obtain an integral packing.

More precisely, we guess $\ell_{\max}$, the index of the highest value class that belongs to Opt and use dynamic linear grouping with $\mathcal{J}' = \mathcal{J}$ and $n' = n$ to obtain $\mathcal{T}$, the set of item types t with their multiplicities n_t, by trying out every $\ell_{\max} \in \{0, \dots, \log_{1+\varepsilon} v_{\max}\}$

Based on $\mathcal{T}$, we group the knapsacks such that any item type is either big or small with respect to every knapsack in a group or does not fit at all. Recall that an item j is small with respect to a knapsack with capacity S_i if $s_j < \varepsilon S_i$ and big otherwise. Hence, we consider the knapsacks sorted non-increasingly by their capacity and determine for each item type for which knapsacks a corresponding item would be big or small. This yields a set $\mathcal{G}$ of $\mathcal{O}\left(\frac{\log^2 n}{\varepsilon^4}\right)$ knapsack groups. In Figure 7.5, we show an example with 4 item types and the resulting knapsack groups.

Denote by $\mathcal{F}_g$ the set of all item types that are small with respect to group g, and by S_g the total capacity of all knapsacks in group g. Let m_g be the number of knapsacks in group g and let $\mathcal{G}^{(1/\varepsilon)}$ be the groups in $\mathcal{G}$ with $m_g \geq \frac{1}{\varepsilon}$. For each $g \in \mathcal{G}^{(1/\varepsilon)}$, define $S_{g,\varepsilon}$ as the total capacity of the smallest $\lceil \varepsilon m_g \rceil$ knapsacks in g. Similar to the ILP for identical knapsacks, the ILP reserves some knapsacks to pack small cut items. We distinguish between $\mathcal{G}^{(1/\varepsilon)}$ and $\mathcal{G} \setminus \mathcal{G}^{(1/\varepsilon)}$ to restrict only large enough groups g, i.e, $g \in \mathcal{G}^{(1/\varepsilon)}$, to $\lfloor (1-\varepsilon) m_g \rfloor$ most valuable knapsacks of g. Per remaining group, we use one knapsack given by resource augmentation to pack cut small items.

For each group $g \in \mathcal{G}$, create all possible configurations of big items that fit into at least one knapsack in group g and therefore consist of at most $\frac{1}{\varepsilon}$ items which are big with respect

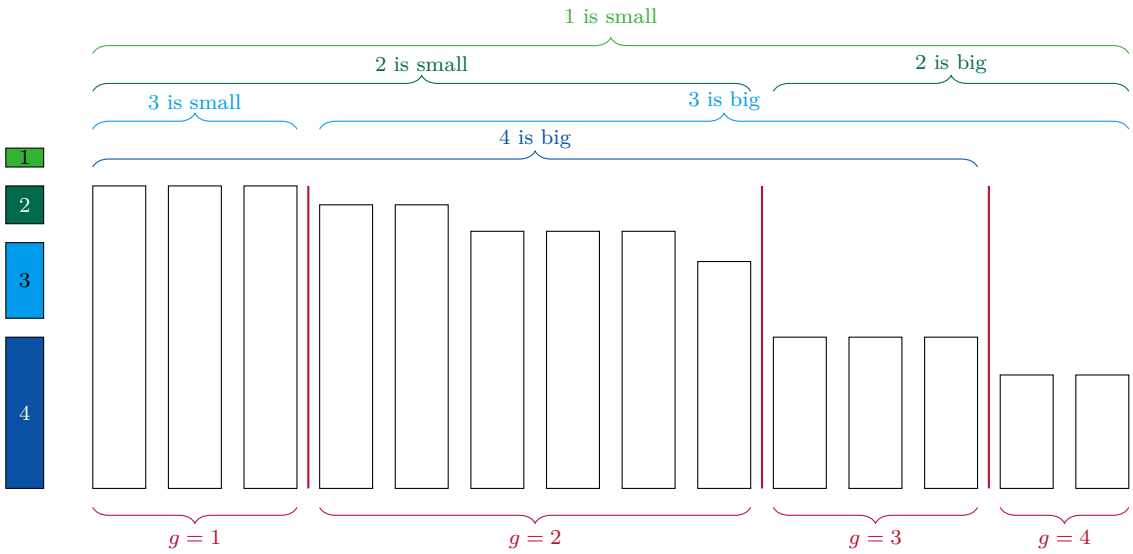

Figure 7.5: Item types and resulting knapsack groups

to knapsacks in g. This amounts to $O\left(\left(\frac{\log^2 n}{\varepsilon^4}\right)^{1/\varepsilon}\right)$ configurations per group. Order the configurations non-increasingly by size and denote their set by $\mathcal{C}_g = \{c_{g,1}, \ldots, c_{g,k_g}\}$. Let $m_{g,\ell}$ be the total number of knapsacks in group g in which we could possibly place configuration $c_{g,\ell}$. Further, denote by $n_{c,t}$ the number of items of type t in configuration c and by s_c and v_c the size and value of c, respectively.

Then, we solve the following configuration ILP with variables y_c and $z_{g,t}$. Here, y_c counts how often a certain configuration c is used, and $z_{g,t}$ counts how many items of type t are packed in knapsacks of group g if type t is small with respect to g. Note that by the above definition of $\mathcal{C}_g$, we may have duplicates of the same configuration for several groups.

$$
\begin{aligned}
\max \quad & \sum_{g\in\mathcal{G}} \sum_{c\in\mathcal{C}_g} y_c v_c + \sum_{g\in\mathcal{G}} \sum_{t\in\mathcal{F}_g} z_{g,t} v_t & & & \\
\text{s.t.} \quad & \sum_{h=1}^{\ell} y_{c_{g,h}} & \leq\ & m_{g,\ell} & \text{for all } g \in \mathcal{G}, \ell \in [k_g] \\
& \sum_{c\in\mathcal{C}_g} y_c & \leq\ & \lfloor (1-\varepsilon) m_g \rfloor & \text{for all } g \in \mathcal{G}^{(1/\varepsilon)} \\
& \sum_{c\in\mathcal{C}_g} y_c s_{c_{g,h}} + \sum_{t\in\mathcal{F}_g} z_{g,t} s_t & \leq\ & S_g & \text{for all } g \in \mathcal{G} \setminus \mathcal{G}^{(1/\varepsilon)} \\
& \sum_{c\in\mathcal{C}_g} y_c s_{c_{g,h}} + \sum_{t\in\mathcal{F}_g} z_{g,t} s_t & \leq\ & S_g - S_{g,\varepsilon} & \text{for all } g \in \mathcal{G}^{(1/\varepsilon)} \\
& \sum_{g\in\mathcal{G}} \sum_{c\in\mathcal{C}_g} y_c n_{c,t} + \sum_{g\in\mathcal{G}: t\in\mathcal{F}_g} z_{g,t} & \leq\ & n_t & \text{for all } t \in \mathcal{T} \\
& y_c & \in\ & \mathbb{Z}^+ & \text{for all } g \in \mathcal{G}, c \in \mathcal{C}_g \\
& z_{g,t} & \in\ & \mathbb{Z}^+ & \text{for all } t \in \mathcal{T}, g \in \mathcal{G} \\
& z_{g,t} & =\ & 0 & \text{for all } t \in \mathcal{T}, g \in \mathcal{G} : t \notin \mathcal{F}_g
\end{aligned}
\tag{P}
$$

The first inequality ensures that the configurations chosen by the ILP actually fit into the knapsacks of the respective group while the second inequality ensures that an ε-fraction of knapsacks in $\mathcal{G}_{1/\varepsilon}$ remains empty for packing small cut items. The third and fourth inequality guarantee that the total volume of large and small items together fits within the designated total capacity of each group. Finally, the fifth inequality makes sure that only available items are used by the ILP.

After relaxing the above ILP and allowing fractional solutions, we are able to solve it efficiently. Consider an optimal (fractional) solution to (P) with objective function value v_{LP}. With Lemma 7.16 we obtain an integral solution that uses the additional knapsacks given by resource augmentation with value at least v_{LP}. Let P denote this final solution.

Still, the small item types $t \in \mathcal{F}_g$ are only packed fractionally by P. Lemma 7.18 explains how to pack the small items integrally. That is, we greedily fill up knapsacks with small items and pack any cut small item into the knapsacks that were left empty by the configuration ILP (or that are provided by the resource augmentation).

We use the solution corresponding to a guess $\ell_{\max}$ that maximizes the total value of packed items. We summarize the algorithm in Algorithm 7.5. Figure 7.6 shows a possible solution.

Algorithm 7.5: Dynamic algorithm for arbitrary knapsacks with resource augmentation

guess $\ell_{\max}$, the largest index of a value class with items in Opt
  use dynamic linear grouping to obtain $\mathcal{T}$
  partition the knapsacks according to $\mathcal{T}$
  solve (P) for $\mathcal{T}$
  use Next Fit to pack the small items per group

Queries Since we do not maintain an explicit packing of any item, we define and update pointers for each item type that dictate the knapsacks where the corresponding items are packed. We note that special pointers are also used for packing items into the additional knapsacks given by resource augmentation. To stay consistent between two update operations, we cache query answers for the current round in the data structure that store items. We give the details in the next section.

- **Single Item Query:** For a queried item, we retrieve its item type and check if it belongs to the smallest items of this type that our implicit solution packs. In this case, we use the pointer for this item type to determine its knapsack.
- **Solution Value Query:** After having found the current solution, we use prefix computation for every value class for the corresponding item types to calculate and store the actual solution value. Then, we return this value on query.
- **Entire Solution Query:** With prefix computation on each value class, we determine the packed items. Then, the single item query is used to determine their knapsack.

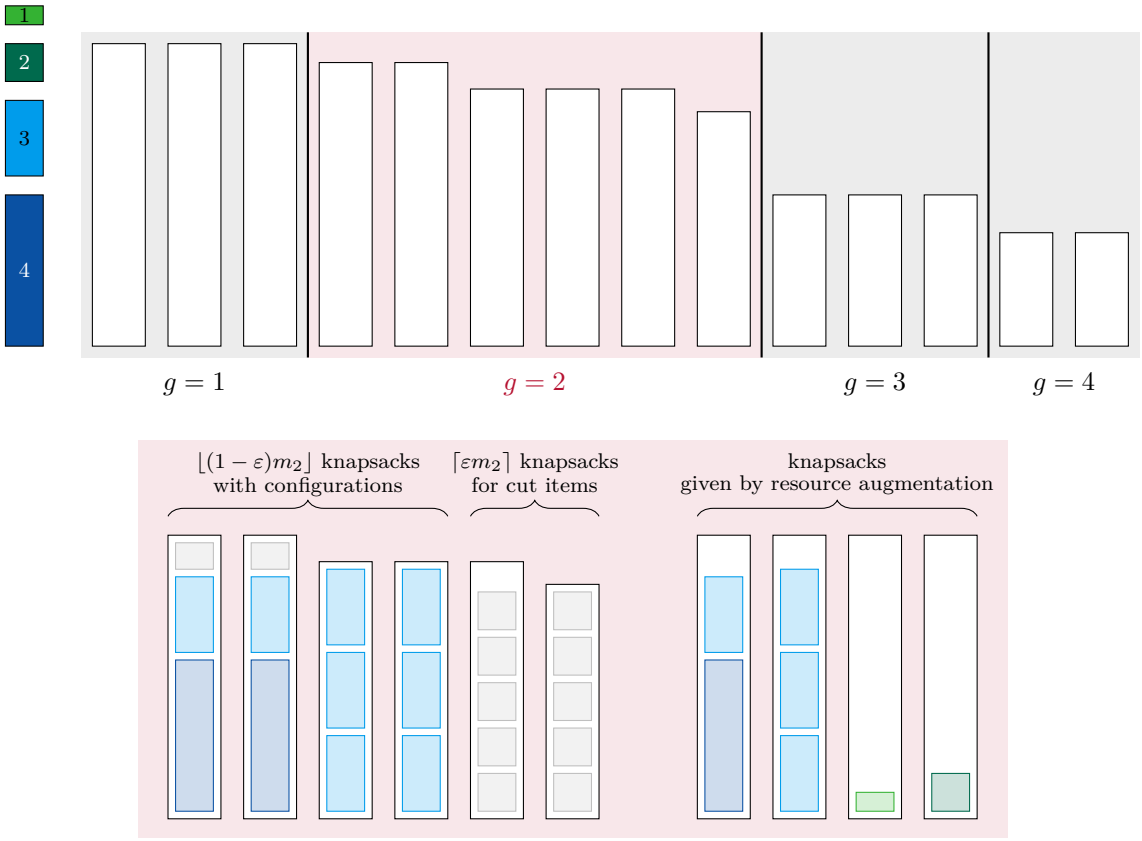

Figure 7.6: Possible solution of the algorithm: Group 2 accommodates the knapsacks for cut small items within the original knapsacks. Group 1, 3, and 4 use resource augmentation.

7.5.2 Analysis

We start again by showing that the loss in the objective function value due to the linear grouping of items is bounded by a factor of at most $\frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2}$ with respect to $v(\text{OPT})$. To this end, let OPT be an optimal solution to the current, non-modified instance and let $\mathcal{J}$ be the set of items with values already rounded to powers of $(1+\varepsilon)$. Setting $\mathcal{J}' = \mathcal{J}$, we apply Theorem 7.4 to obtain the following corollary. Here, $\text{OPT}_{\mathcal{T}}$ is a optimal solution for the instance induced by the item types $\mathcal{T}$ with multiplicities n_t.

Corollary 7.28. *There exists an index $\ell_{\max}$ such that $v(\text{OPT}_{\mathcal{T}}) \geq \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2} v(\text{OPT})$.*

We have thus justified the restriction to item types $\mathcal{T}$ instead of packing the actual items. In the next two lemmas, we show that (P) is a linear programming formulation of the MULTIPLE KNAPSACK problem described by the set $\mathcal{T}$ of item types and their multiplicities and that we can obtain a feasible integral packing (using resource augmentation) if we have a fractional solution (without resource augmentation) to (P). Let v_{LP} be the optimal objective function value of the LP relaxation of (P).

Similar to the proof of Lemma 7.13 we restrict an optimal solution $\text{OPT}_{\mathcal{T}}$ to the $\lfloor(1-\varepsilon)m_g\rfloor$ most valuable knapsacks of a group g if $m_g \geq \frac{1}{\varepsilon}$ and otherwise we do not restrict the part of the solution corresponding to a group g with $m_g < \frac{1}{\varepsilon}$.

Lemma 7.29. *It holds that* $v_{LP} \geq (1-2\varepsilon)v(\text{OPT}_{\mathcal{T}})$.

Proof. We show the statement by explicitly stating a solution (y, z) that is feasible for (P) and achieves an objective function value of at least $(1-2\varepsilon)v(\text{OPT}_{\mathcal{T}})$.

Consider a feasible optimal packing $\text{OPT}_{\mathcal{T}}$ for item types. The construction of (y, z) considers each group $g \in \mathcal{G}$ separately. We fix a group $g \notin \mathcal{G}^{(1/\varepsilon)}$. Let y_c count how often a configuration $c \in \mathcal{C}_g$ is used in $\text{OPT}_{\mathcal{T}}$ and let $z_{g,t}$ denote how often an item that is small with respect to g is packed by $\text{OPT}_{\mathcal{T}}$ in group g. By construction, the first and the third constraint of (P) are satisfied. The part of the solution (y, z) corresponding to group g achieves the same value as $\text{OPT}_{\mathcal{T}}$ restricted to this group.

If $g \in \mathcal{G}^{(1/\varepsilon)}$, i.e., if there are at least $\frac{1}{\varepsilon}$ knapsacks in group g, consider the $\lfloor(1-\varepsilon)m_g\rfloor$ most valuable knapsacks in group g with respect to $\text{OPT}_{\mathcal{T}}$. Define y_c to count how often $\text{OPT}_{\mathcal{T}}$ uses configuration $c \in \mathcal{C}_c$ in this reduced knapsack set and let $z_{g,t}$ denote how often $\text{OPT}_{\mathcal{T}}$ uses item type $t \in \mathcal{F}_g$ in these knapsacks. Clearly, this solution satisfies the first constraint of (P). By construction, $\sum_{c\in\mathcal{C}_g} y_c \leq \lfloor(1-\varepsilon)m_g\rfloor$ and, hence, the second constraint of the ILP is also satisfied. Clearly, the $\lfloor(1-\varepsilon)m_g\rfloor$ most valuable knapsacks can be packed into the $\lfloor(1-\varepsilon)m_g\rfloor$ largest knapsacks in g, which implies the feasibility for the fourth constraint of the ILP. Observe that $\lfloor(1-\varepsilon)m_g\rfloor \geq (1-\varepsilon)m_g - 1 \geq (1-2\varepsilon)m_g$. Thus, the value of the corresponding packing is at least a $(1-2\varepsilon)$-fraction of the value that $\text{OPT}_{\mathcal{T}}$ obtains with group g.

As (y, z) uses no more items of a certain item type than $\text{OPT}_{\mathcal{T}}$ does, the last constraint of the ILP is also satisfied. Hence, (y, z) is feasible and

$$v_{\text{LP}} \geq \sum_{g\in\mathcal{G}} \left(\sum_{c\in\mathcal{C}_g} y_c v_c + \sum_{t\in\mathcal{F}_g} z_{g,t} v_t \right) \geq (1-2\varepsilon)v(\text{OPT}_{\mathcal{T}}),$$

with which we conclude the proof. □

The next corollary shows how to round any fractional solution of (P) to an integral solution (possibly) using additional knapsacks given by resource augmentation. It follows immediately from Lemma 7.16 if we bound the number of variables in (P). To this end, we observe that $|\mathcal{G}|$ and $|\mathcal{T}|$ are in $\mathcal{O}\left(\frac{\log^2 n}{\varepsilon^4}\right)$, and $|\mathcal{C}_g| \in \left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}$ for every group $g \in \mathcal{G}$. Let L' denote the exact number of variables and let $L = L' + |\mathcal{G}|$. Thus, $L \in \left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}$.

Corollary 7.30. *Any feasible solution (y, z) of the LP relaxation of (P) with objective value v can be rounded to an integral solution with value at least v using at most L extra knapsacks.*

In the next lemma, we bound the value obtained by our algorithm in terms of $v(\textsc{Opt})$, for an optimal solution $\textsc{Opt}$. Let P_F be the solution returned by our algorithm.

Lemma 7.31. $v(P_F) \geq \frac{(1-2\varepsilon)^2(1-\varepsilon)}{(1+\varepsilon)^2} v(\textsc{Opt})$.

Proof. Fix an optimal solution $\textsc{Opt}$. Observe that our algorithm outputs the solution P_F with the maximum value over all guesses of $\ell_{\max}$, the index of the highest value class in $\textsc{Opt}$. Hence, we find a guess $\ell_{\max}$ and a corresponding solution P that satisfies $v(P) \geq \frac{(1-2\varepsilon)^2(1-\varepsilon)}{(1+\varepsilon)^2} v(\textsc{Opt})$.

Let $\ell_{\max} = \max\{\ell : V_\ell \cap \textsc{Opt} \neq \emptyset\}$. Then, $\ell_{\max}$ is considered in some round of the algorithm. Let v_{ILP} be the optimal solution value of the configuration ILP (P) and let v_{LP} be the solution value of its LP relaxation. Corollary 7.30 provides a way to round the corresponding LP solution (y, z) to an integral solution $(\bar{y}, \bar{z})$ using at most L extra knapsacks with objective function value at least $v_{\text{LP}} \geq v_{\text{ILP}}$. The construction of $(\bar{y}, \bar{z})$ guarantees that only small items in the original knapsacks might be packed fractionally.

Consider one particular group g. Lemma 7.18 shows how to pack the small items assigned by $(\bar{z}_g)$ to group g into $\lceil (1+\varepsilon) m_g \rceil$ knapsacks. If $m_g < \frac{1}{\varepsilon}$, we use one extra knapsack per group to pack the cut items. If $m_g \geq \frac{1}{\varepsilon}$, then $g \in \mathcal{G}^{(1/\varepsilon)}$ which implies that the configuration ILP (and its relaxation) already reserved $\lceil \varepsilon m_g \rceil$ knapsacks of this group for packing small items. Hence, the just obtained packing P is feasible. By Corollary 7.28 and Lemma 7.29,

$$v(P_F) \geq v(P) \geq \frac{(1-2\varepsilon)^2(1-\varepsilon)}{(1+\varepsilon)^2} v(\textsc{Opt}),$$

which gives the desired bound on the approximation ratio. □

Now, we bound the running time of our algorithm.

Lemma 7.32. *In time* $\left(\frac{1}{\varepsilon} \log n\right)^{\mathcal{O}(1/\varepsilon)} (\log m \log S_{\max} \log v_{\max})^{\mathcal{O}(1)}$, *the dynamic algorithm executes one update operation.*

Proof. By assumption, upon arrival, the value of each item is rounded to natural powers of $(1+\varepsilon)$. The algorithm starts with guessing $\ell_{\max}$, the largest index of a value class to be considered in the current iteration. There are $\log v_{\max}$ many guesses possible, where $v_{\max}$ is the highest value appearing in the current instance.

By Lemma 7.10, the dynamic linear grouping of all items has at most $\mathcal{O}\left(\frac{\log^4 n}{\varepsilon^4}\right)$ iterations.

Let the knapsacks be sorted by increasing capacity and stored in a binary balanced search tree as defined in Lemma 7.2. Then, the index of the smallest knapsack i with $S_i \geq S$ or the largest knapsack with $S_i \leq S$ can be determined in time $\mathcal{O}(\log m)$, where S is a given number. Thus, the knapsack groups depending on the item types can be determined in time $\mathcal{O}\left(\log m \frac{\log^2 n}{\varepsilon^4}\right)$ as the number of item types is bounded by $\mathcal{O}\left(\frac{\log^2 n}{\varepsilon^4}\right)$. The number of big items per knapsack is bounded by $\frac{1}{\varepsilon}$ and, hence, the number of configurations is bounded by $\mathcal{O}\left(\frac{\log^2 n}{\varepsilon^4}\left(\frac{\log^2 n}{\varepsilon^4}\right)^{1/\varepsilon}\right)$.

Let N be the number of variables in the configuration ILP. We have $N \in \left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}$. Hence, there is a polynomial function $g(N, \log S_{\max}, \log v_{\max})$ that bounds the running time of finding an optimal solution to the LP relaxation of the configuration ILP [BT97,PS82]. Clearly, the computational complexity of setting up and rounding the fractional solution is dominated by solving the LP. Thus, $\left(\frac{1}{\varepsilon}\log n\right)^{\mathcal{O}(1/\varepsilon)}(\log m \log S_{\max} \log v_{\max})^{\mathcal{O}(1)}$ bounds the running time.

In similar time, we can store y and z, the obtained solutions to the configuration LP. Let $\bar{y}$ and $\bar{z}$ be the variables obtained by (possibly) rounding down y and z and let $\tilde{y}$ and $\tilde{z}$ be the variables corresponding to the resource augmentation as in Lemma 7.16. The time needed to obtain these variables is dominated by solving the LP relaxation of the configuration ILP. □

Answering queries Since we only store implicit solutions, it remains to show how to answer the corresponding queries. In order to determine the relevant parameters of a particular item, we assume that all items are stored in one balanced binary search tree that allows us to access one item in time $\mathcal{O}(\log n)$ by Lemma 7.2. We additionally assume that this balanced binary search tree also stores the value class of an item. We use again the round parameter $t(j)$ and the corresponding knapsack $k(j)$ to cache given answers in order to stay consistent between two updates. If j was NOT SELECTED in round $t(j)$, we represent this by $k(j) = 0$. We assume that these two parameters are stored in the same binary search tree that also stores the items and, thus, can be accessed in time $\mathcal{O}(\log n)$.

We now design an algorithm for non-cached items. The high-level idea is similar to the algorithm developed in Section 7.4 for identical knapsacks. As the knapsacks have different capacities in this section, the relative size of an item depends on the particular knapsack group: An item can be big with respect to one knapsack and small with respect to another. Thus, the distinction between small and big items does not hold for all knapsacks simultaneously anymore and needs to be handled carefully. More precisely, upon query of an item j of type t, we start by determining the group γ_t in which the next item of type t is packed. The pointers and counters we use correspond mostly to the ones in Section 7.4 except that we additionally have a dependency on the particular group g for each parameter. Additionally, we use $R_g^{(\varepsilon)}$, $R_g^{(y)}$ and $R_g^{(z)}$ to refer to knapsacks given by resource augmentation for group g.

If t is small with respect to γ_t, then j is packed by NEXT FIT either as regular or as cut item. We use the two pointers κ_g^r for packing small items regularly in group g and κ_g^c for packing cut items. If there are at most $\frac{1}{\varepsilon} - 1$ knapsacks in group g, then κ_g^c points to the knapsack $R_g^{(\varepsilon)}$ given by resource augmentation. Otherwise, the configuration ILP left the smallest $\lceil \varepsilon m_g \rceil$ knapsacks in group g empty for packing cut small items. Further, we use $R_{g,t}^{(z)}$ to refer to the knapsack given by resource augmentation that is used for packing one item of type t if the variable $z_{g,t}$ was subjected to rounding. Since we may only pack as many items of type t in group g as indicated by the implicit solution, the counter η_t^S determines how many items of type t can still be packed in group γ_t if t is small with respect to γ_t.

If t is big with respect to γ_t, then j is packed in the next slot for items of type t determined by the configuration ILP. To this end, we use again the counter κ_t to determine the knapsack where the next item of type t is packed and the counter η_t^B to determine how many items of type t can still be packed in knapsack κ_t if t is big with respect to γ_t. The knapsack $R_{c,g}^{(y)}$, for $c \in \mathcal{C}_g$, refers to the knapsack given by resource augmentation used when the variable $y_{c,g}$ was subjected to rounding.

Table 7.1 summarizes the parameters and counters used to answer queries, and in Figure 7.7, we give an example of the current packing after some items have been queried. Next, we define the data structures for answering queries before we formally explain how to answer queries.

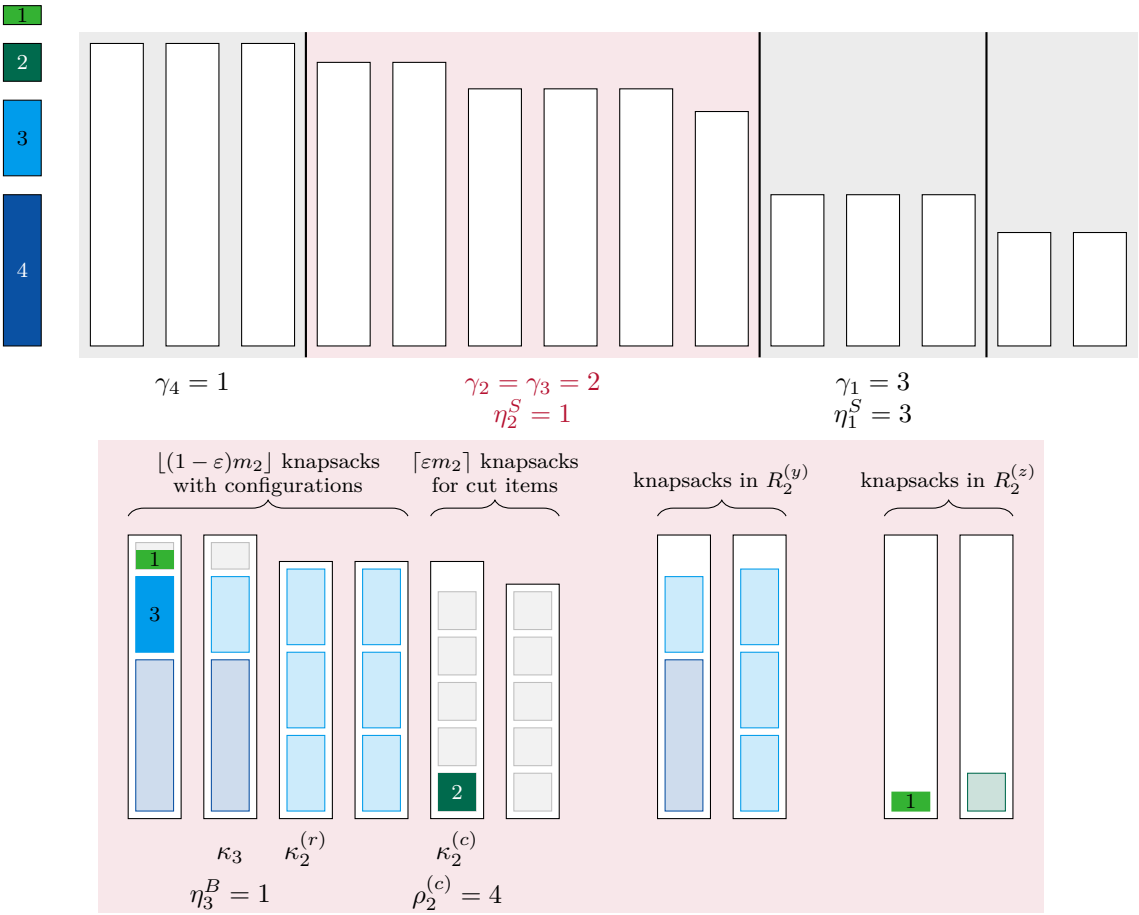

Figure 7.7: Counters and pointers for answering queries: Gray rectangles inside knapsacks represent small items. The next item of type 2 (dark green) is placed in the knapsack given by resource augmentation $R_2^{(z)}$ since $\eta_2^S = \tilde{z}_{2,2}$. Items of type 1 (light green) already filled all their slots in group 2 and are now placed in group 3.

Data structures We assume that the knapsacks are sorted by non-increasing capacity and stored in one binary search tree together with S_i, the capacity of the knapsacks. The knapsacks

Table 7.1: Counters and pointers used during querying items

Counter/Pointer	Meaning
$\overline{\mathcal{C}}_g$	Configurations that are used by group g
$\alpha_{c,g}$	First knapsack with configuration c in group g
$R^{(y)}_{c,g}$	Knapsack in $R^{(y)}$ used for group g and configuration c
$R^{(z)}_{g,t}$	Knapsack in $R^{(z)}$ used for group g and type t
$R^{(\varepsilon)}_g$	Knapsack in $R^{(\varepsilon)}$ used for group g with $m_g < \frac{1}{\varepsilon}$
$\mathcal{G}_t$	Knapsack groups where items of type t are packed
$\mathcal{C}_{g,t}$	List of configurations $c \in \overline{\mathcal{C}}_g$ with $n_{c,t} \geq 1$
γ_t	Current knapsack group where items of type t are packed
κ_t	Current knapsack for packing items of a big type t
η^S_t	Remaining number of slots for items of type t in γ_t
η^B_t	Remaining number of slots for items of type t in κ_t
κ^r_g	Current knapsack in g for packing small items regularly
κ^c_g	Current knapsack in g (or in $R^{(\varepsilon)}$) for packing cut small items
ρ^r_g	Remaining capacity in κ^r_g for packing small items
ρ^c_g	Remaining number of slots for small items in κ^c_g

given by resource augmentation are stored in three different lists, $R^{(y)}$, $R^{(z)}$, and $R^{(\varepsilon)}$, needed due to rounding y or z or because $m_g < \frac{1}{\varepsilon}$, respectively. The knapsack groups are stored in the list $\mathcal{G}$ sorted by non-increasing knapsack capacity. For each group g, we additionally store the number m_g of knapsacks belonging to g.

Let $\bar{y}, \tilde{y}, \bar{z}$, and $\tilde{z}$ be the implicit solution of the algorithm. Here $\bar{*}$ refers to packing configurations or items into the original knapsacks while $\tilde{*}$ refers to the knapsacks given by resource augmentation. Let $\overline{\mathcal{C}}_g$ be the set of configurations c with $\bar{y}_{c,g}+\tilde{y}_{c,g} \geq 1$ ordered in non-increasing size s_c and stored in one list per group. In the following, we use the position of a configuration $c \in \overline{\mathcal{C}}_g$ in that list as the index of c. For mapping the configurations to knapsacks, we assign the knapsacks $\sum_{g'=1}^{g-1} m_{g'} + \sum_{c'=1}^{c-1} \bar{y}_{c',g} + 1, \ldots, \sum_{g'=1}^{g-1} m_{g'} + \sum_{c'=1}^{c} \bar{y}_{c',g}$ to configuration c. For the knapsacks in the resource augmentation, we set $R^{(y)}_{c,g} = \sum_{g'=1}^{g-1} \sum_{c' \in \overline{\mathcal{C}}_{g'}} \tilde{y}_{c',g'} + \sum_{c' \leq c} \tilde{y}_{c',g}$ for each group g and each configuration $c \in \overline{\mathcal{C}}_g$.

For each item type t, let $\bar{n}_t$ denote the number of items of type t in the solution. We maintain a pointer γ_t to the group where the next queried item of type t is supposed to go. We initialize γ_t with the first group that packs items of type t. Since the number of items of type t assigned to group g as small items is determined by $\bar{z}_{g,t} + \tilde{z}_{g,t}$, we additionally use the counter η^S_t, initialized with $\bar{z}_{\gamma_t,t} + \tilde{z}_{\gamma_t,t}$, to reflect how many slots group γ_t still has for items of type t. For accessing the knapsacks $R^{(z)}$ given by resource augmentation, we set $R^{(z)}_{g,t} = \sum_{g'=1}^{g-1} \sum_{t' \in \mathcal{T}} \tilde{z}_{g',t'} + \sum_{t'=1}^{t} \tilde{z}_{g,t'}$ for each group g and item type t. Note that $z_{g,t} = 0$ holds if t is big with respect to g.

When packing small items in group g, we use group pointers κ^r_g and κ^c_g to refer to the knapsack for packing items regularly or for packing cut items. The pointer κ^r_g is initialized

with $\kappa_g^r = \sum_{g'=1}^{g-1} m_{g'} + 1$. Further, we use ρ_g^r to store the remaining capacity for small items in κ_g^r and initialize it with $\rho_g^r = S_{\kappa_g^r} - s_1$, where s_1 is the size of the first configuration in group g. If $m_g \geq \frac{1}{\varepsilon}$, we set $\kappa_g^c = \sum_{g'=1}^{g-1} m_{g'} + \lfloor (1-\varepsilon)m_g \rfloor + 1$, while $m_g < \frac{1}{\varepsilon}$ implies that κ_g^c points to the knapsack $R_g^{(\varepsilon)}$ given by resource augmentation. The counter ρ_g^c stores again the remaining slots for cut small items in group g and is initialized with $\frac{1}{\varepsilon}$.

If t is big with respect to γ_t, we use the pointer κ_t to direct us to the particular knapsack where the next item of type t goes, while η_t^B stores how many slots κ_t still has available for items of type t. Initially, κ_t points to the first knapsack with a configuration that contains t in the first group where t is packed as big item. If c is the corresponding configuration, we set $\eta_t^B = n_{c,t}$. Because of resource augmentation, κ_t may point to a knapsack in $R^{(y)}$, the additional knapsacks for rounding y.

Queries Consider a queried, non-cached item j with value class V_ℓ. If $\ell \notin \{\ell_{\min}, \ldots, \ell_{\max}\}$, we return $k(j) = 0$. Otherwise, let t be its type. We check if j belongs to the first $\bar{n}_t$ items of this type. If not, then $k(j) = 0$ is returned. Otherwise, let γ be the group γ_t where the next item of type t is packed.

We first consider the case that t is small with respect to group γ. Recall that η_t^S stores the number of remaining slots of group γ. If $\eta_t^S = \tilde{z}_{\gamma,t}$, then all original slots of group γ are already filled with items of type t. Hence, j either goes to the knapsack $R_{\gamma,t}^{(z)}$ from resource augmentation $R^{(z)}$ or to the next group. If $\tilde{z}_{\gamma,t} = 1$, then j is packed in $k(j) = R_{\gamma,t}^{(z)}$. We update γ_t to point to the next group that packs items of type t and update η_t^S according to the new group if t is still small with respect to γ_t. (Otherwise the next item of type t will be packed according to κ_t.) Then, we return $k(j)$. Else, we update γ_t to point to the next group that packs items of type t and update η_t^S accordingly if t is still small with respect to γ_t. Then, the case distinction on the size of t relative to γ_t is invoked again. If $\eta_t > \tilde{z}_{\gamma,t}$, then we decrease η_t by one and pack j among the original knapsacks. We need to determine if j is packed regularly or as a cut item. To this end, we compare s_j with ρ_γ^r. If $s_j \leq \rho_\gamma^r$, we pack j in knapsack κ_γ^r. Next, ρ_γ^r is decreased by s_j, and we return $k(j) = \kappa_\gamma^r$. Otherwise, we close knapsack κ_γ^r for small items by increasing this pointer by one and pack j as cut item, i.e., $k(j) = \kappa_\gamma^c$. We reflect this decision by decreasing ρ_γ^c by one. If this leads to $\rho_\gamma^c = 0$, we increase κ_γ^c by one and set $\rho_\gamma^c = \frac{1}{\varepsilon}$. Further, we update ρ_g^r.

Now consider the case where t is big with respect to group γ. Then, the pointer κ_t dictates the knapsack of j. We decrease η_t^B by one. If this leads to $\eta_t^B = 0$, we find the next knapsack (either in group γ or in the next group) that packs items of type t and update κ_t, η_t^B, and possibly γ_t accordingly. This algorithm is summarized in Algorithm 7.6.

Algorithm 7.6: Answering item queries in round τ

```
if t(j) ≠ τ
    t(j) ← τ
```

```
ℓ ← index of the value class of j
if ℓ ∉ {ℓ_min, ..., ℓ_max} do
    k(j) ← 0 // not selected
else
    t ← type of j, γ ← γ_t
    if j ∈ first η̄_t items do
        if t small w.r.t. γ do // case distinction for big and small items
            if η_t^S = z̃_{γ,t} and z̃_{γ,t} = 1 do // resource augmentation
                k(j) ← R_{g,t}^{(z)}
                update γ_t to the next group
                update η_t^S accordingly or use κ_t and η_t^B if necessary
            else-if η_t^S = z̃_{γ,t} and z̃_{γ,t} = 0 do // next group
                update γ_t to the next group
                update η_t^S accordingly or use κ_t and η_t^B if necessary
                go to back to the case distinction for big and small items
            else-if s_j ≤ ρ_γ^r do // regular item
                k(j) ← κ_γ^r
                η_t^S ← η_t^S − 1
                ρ_γ^r ← ρ_γ^r − s_j
            else // cut item
                k(j) ← κ_γ^c
                η_t^S ← η_t^S − 1
                κ_γ^r ← κ_γ^r + 1
                ρ_γ^c ← ρ_γ^c − 1
                if ρ_γ^c = 0
                    κ_γ^c ← κ_γ^c + 1
                    ρ_γ^c ← 1/ε
        else // big item
            k(j) ← κ_t
            η_t^B ← η_t^B − 1
            update κ_t, η_t^B, and γ_t if necessary
    else
        k(j) ← 0 // not selected
    return k(j)
```

For calculating the value of the current solution, we need to calculate the total value of the first $\bar{n}_t$ items. We do this by iterating through the value classes once and per value class, we iterate once through the list $\mathcal{T}_\ell$ of item types for value class V_ℓ to access the number $\bar{n}_t$. Then, we use prefix computation twice in order to access the total value of the first $\bar{n}_t$ items of type t. Again, we do this computation once after each update operation. Lemma 7.36 bounds the running time of these calculations and shows that incorporating these does not change the order of magnitude of the running time given in Lemma 7.32.

For returning the complete solution, we iterate once through the value classes and for each

value class, we iterate through the list $\mathcal{T}_\ell$ to access the number $\bar{n}_t$. Then, we use prefix computation based on the indices of the items for accessing the first $\bar{n}_t$ items of type t. Then, we access and query each item individually.

We prove the parts of the next lemma again separately.

Lemma 7.33. *The solution determined by the query algorithm is feasible as well as consistent and achieves the claimed total value. The query times of our algorithm are as follows.*

(i) Single item queries can be answered in time $\mathcal{O}\Big(\log m + \frac{\log n}{\varepsilon^2}\Big)$.

(ii) Solution value queries can be answered in time $\mathcal{O}(1)$.

(iii) Queries of the entire solution P *are answered in time* $\mathcal{O}\Big(|P|\frac{\log^3 n}{\varepsilon^4}\Big(\log m + \frac{\log n}{\varepsilon^2}\Big)\Big)$.

Lemma 7.34. *The query algorithms return a feasible and consistent solution obtaining the total value given by the implicit solution.*

Proof. By construction of $k(j)$ and $t(j)$, the solution returned by the query algorithms is consistent between updates.

Observe that $\bar{y}$ and $\bar{z}$ is a feasible solution to the configuration ILP (P). Hence, showing that the algorithm does not assign more than $\bar{y}_{c,g}$ times configuration c and not more than $\bar{z}_{g,t}$ items of type t to group g is sufficient for having a feasible packing of the corresponding elements into the $\lfloor(1-\varepsilon)m_g\rfloor$ largest knapsacks of group g if $m_g \geq \frac{1}{\varepsilon}$ or into the m_g knapsacks of group g if $m_g < \frac{1}{\varepsilon}$. When defining L, we made sure that the items and configurations specified by $\tilde{y}$ and $\tilde{z}$ fit into the knapsacks given by resource augmentation.

If the item type t is small with respect to the group g, then at most $\bar{z}_{g,t}$ items of type t are packed in group g. Thus, Lemma 7.18 ensures that all small items assigned to group g fit in the knapsacks for regular and the cut items. Moreover, the treatment of $\eta_t^S = \tilde{z}_{g,t}$ guarantees that the value obtained by small items packed in g and its additional knapsacks is as in the implicit solution.

If t is big with respect to group g, then the constructions of κ_t and η_t^B ensure that exactly $\sum_{c\in\overline{\mathcal{C}}_g}(\bar{y}_c + \tilde{y}_c)n_{c,t}$ items of type t are packed in group g and in $R_g^{(y)}$. Hence, the total value achieved is as given by the implicit solution. □

Lemma 7.35. *The data structures can be generated in* $\mathcal{O}\Big(\frac{\log^4 n}{\varepsilon^8}\Big(\log m + \frac{\log^{2/\varepsilon} n}{\varepsilon^{4/\varepsilon}}\Big)\Big)$ *many iterations. Queries for a particular item can be answered in* $\mathcal{O}\Big(\log m + \frac{\log n}{\varepsilon^2}\Big)$ *many steps.*

Proof. We start by retracing the steps of the dynamic linear grouping in order to obtain the set $\mathcal{T}$ of item types. We store the types $\mathcal{T}_\ell$ of one value class in one list, sorted by non-decreasing size. By Lemma 7.10, the set $\mathcal{T}$ can be determined in time $\mathcal{O}\Big(\frac{\log^4 n}{\varepsilon^4}\Big)$.

We first argue about the generation of the data structures and the initialization of the various pointers and counters. We start by generating a list $(\alpha_{c,g})_{c\in\overline{\mathcal{C}}_g}$ for each group g where $\alpha_{c,g}$ stores the first (original) knapsack of configuration $c \in \overline{\mathcal{C}}_g$, i.e.,

$$\alpha_{c,g} = \alpha_{c-1,g} + \bar{y}_{c-1,g} + 1,$$

where $\alpha_{0,g} = \sum_{g'=1}^{g-1} m_{g'}$ and $y_{0,g} = 0$. Next, we set

$$R_{g,t}^{(z)} = \sum_{g'=1}^{g-1} \sum_{t' \in \mathcal{T}} \tilde{z}_{g',t'} + \sum_{t'=1}^{t} \tilde{z}_{g,t'}$$

and

$$R_{c,g}^{(y)} = \sum_{g'=1}^{g-1} \sum_{c' \in \overline{\mathcal{C}}_{g'}} \tilde{y}_{c',g'} + \sum_{c' \in \mathcal{C}_g, c' \leq c} \tilde{y}_{c',g}\,,$$

where $R_{g,t}^{(z)}$ corresponds to the resource augmentation needed because of rounding $z_{g,t}$ and $R_{c,g}^{(y)}$ corresponds to the resource augmentation for rounding $y_{c,g}$. These lists can be generated by iterating through the list $\overline{\mathcal{C}}_g$ for each group g in time $\mathcal{O}(\sum_{g \in \mathcal{G}} |\mathcal{C}_g|) = \mathcal{O}\Big(\frac{\log^2 n}{\varepsilon^4} \frac{\log^{2/\varepsilon} n}{\varepsilon^{4/\varepsilon}}\Big)$.

For maintaining and updating the pointer γ_t, we generate the list $\mathcal{G}_t$ that contains all groups g where items of type t are packed in the implicit solution. By iterating through the groups once more and checking $\sum_{c \in \overline{\mathcal{C}}_g} (\bar{y}_{c,g} + \tilde{y}_{c,g}) n_{c,t} \geq 1$ or $\bar{z}_{g,t} + \tilde{z}_{g,t} \geq 1$, we can add the corresponding groups g to $\mathcal{G}_t$. Then, γ_t points to the head of the list. While iterating through the groups, we also calculate $\bar{n}_t = \sum_{g \in \mathcal{G}} \Big(\sum_{c \in \mathcal{C}'_g} (\bar{y}_{c,g} + \tilde{y}_{c,g}) + \bar{z}_{g,t} + \tilde{z}_{g,t} \Big)$ and store the corresponding value together with the item type. The lists $\mathcal{G}_t$ can be generated in $\mathcal{O}(|\mathcal{T}| \sum_{g \in \mathcal{G}} |\mathcal{C}_g|) = \mathcal{O}\Big(\frac{\log^4 n}{\varepsilon^8} \frac{\log^{2/\varepsilon} n}{\varepsilon^{4/\varepsilon}}\Big)$ many iterations.

For maintaining and updating the pointer κ_t we create the list $\mathcal{C}_{g,t}$ storing all configurations $c \in \overline{\mathcal{C}}_g$ with $n_{c,t} \geq 1$. While iterating through the groups and creating $\mathcal{G}_t$, we also add c together with $n_{c,t}$ to the list $\mathcal{C}_{g,t}$ if $n_{c,t} \geq 1$. Initially, κ_t points to the head of $\mathcal{C}_{g,t}$, where g is the first group that packs t as big item. If c is the corresponding configuration, we start with $\eta_t^B = n_{c,t}$. The time needed for this is bounded by $\mathcal{O}(|\mathcal{T}| \sum_{g \in \mathcal{G}} |\mathcal{C}_g|) = \mathcal{O}\Big(\frac{\log^4 n}{\varepsilon^8} \frac{\log^{2/\varepsilon} n}{\varepsilon^{4/\varepsilon}}\Big)$.

The pointer κ_g^r is initialized with $\kappa_g^r = \sum_{g'=1}^{g-1} m_{g'} + 1$. By using binary search on the list $\overline{\mathcal{C}}_g$, we get s_1, the total size of configuration 1 assigned to κ_g^r, and binary search over the knapsacks allows us to obtain $S_{\kappa_g^r}$, the capacity of knapsack κ_g^r. Thus, $\rho_g^r = S_{\kappa_g^r} - s_1$ can be initialized in time $\mathcal{O}(\sum_{g \in \mathcal{G}} (\log(|\overline{\mathcal{C}}_g| + \log m)) = \mathcal{O}\Big(\frac{\log^2 n}{\varepsilon^5} \Big(\log \frac{\log n}{\varepsilon} + \log m \Big)\Big)$.

If $m_g \geq \frac{1}{\varepsilon}$, we set $\kappa_g^c = \sum_{g'=1}^{g-1} m_{g'} + \lfloor (1-\varepsilon) m_g \rfloor + 1$, while $m_g < \frac{1}{\varepsilon}$ implies that κ_g^c points to the knapsack $R^{(\varepsilon)} = |\{g' : g' \leq g, m_{g'} < \frac{1}{\varepsilon}\}|$ given by resource augmentation. The time needed for initializing κ_g^c is $\mathcal{O}(|\mathcal{G}|)$. In order to determine the position of the next cut item, we also maintain ρ^c, initialized with $\rho_g^c = \frac{1}{\varepsilon}$, that counts how many slots are still left in knapsack κ_g^c.

Now consider the query for an item j. We can decide in time $\mathcal{O}(\log n)$ if j has already been queried in the current round. Upon arrival of j, we calculated the index ℓ of its value class. If $\ell \in \{\ell_{\min}, \ldots, \ell_{\max}\}$, then the item types $\mathcal{T}_\ell$ together with their first and last item can be determined in time $\mathcal{O}\Big(\frac{\log n}{\varepsilon^2}\Big)$ by retracing the steps of the linear grouping,. By binary search, the item type of j can be determined in time $\mathcal{O}\Big(\log \frac{\log n}{\varepsilon}\Big)$. Once the item type is known, we check if j belongs to the first $\bar{n}_t$ items of this type. If not, then NOT SELECTED is returned.

Otherwise, the pointer γ_t answers the question in which group item j is packed.

If j is small and $\eta_t^S > \tilde{z}_{t,\gamma_t}$, the knapsack $k(j)$ can be determined in constant time by nested case distinction and having the correct pointer (either $\kappa^r_{\gamma_t}$ or $\kappa^c_{\gamma_t}$) dictate the answer. In order to bound the update time of the data structures, note that packing j as regular item only implies the updates of ρ_{γ_t} and of η_t^S, which take constant time. Hence, it remains to consider the case where j is packed as a cut item. The capacity of the new knapsack $\kappa^r_{\gamma_t}$ can be determined in $\mathcal{O}(\log m)$ by binary search over the knapsack list while the configuration c of the new knapsack $\kappa^r_{\gamma_t}$ and its total size are determined by binary search over the list α_{γ_t} in time $\mathcal{O}\left(\log|\overline{\mathcal{C}}_{\gamma_t}|\right) = \mathcal{O}\left(\frac{1}{\varepsilon}\log\frac{\log n}{\varepsilon}\right)$. Then, $\rho^r_{\gamma_t} = S_{\kappa^r_{\gamma_t}} - s_c$ can be computed with constantly many operations. If $\rho^c_{\gamma_t} = 0$ after packing j in $\kappa^c_{\gamma_t}$, we increase the knapsack pointer by one and update $\rho^c_{\gamma_t} = \frac{1}{\varepsilon}$. In case $\eta_t^S = \tilde{z}_{\gamma_t,t} = 1$, item j is packed in the knapsack $R^{(z)}_{\gamma_t,t}$ which can be decided in constant time. Otherwise the group pointer γ_t is increased and either η_t^S is updated according to the new group or κ_t and η_t^B are used. Updating γ_t can be done by binary search over the list $\mathcal{G}_t$ in time $\mathcal{O}(\log|\mathcal{G}|)$. The pointer γ_t is updated at most once *before* determining $k(j)$. Hence, the case distinction on the relative size of t is invoked at most twice.

If j is big, the pointer κ_{γ_t} dictates the answer which can be returned in time $\mathcal{O}(1)$. For bounding the running time of the possible update operations, observe that η_t is updated in constant time with values bounded by n. If $\eta_t^B = 0$ after the update, the knapsack pointer κ_t needs to be updated as well. The most time consuming update operations are finding a new configuration c' and possibly even a new group g'. Finding configuration $c' \in \overline{\mathcal{C}}_{\gamma_t,t}$ can be done by binary search in time $\mathcal{O}\left(\log|\overline{\mathcal{C}}_{\gamma_t}|\right) = \mathcal{O}\left(\frac{1}{\varepsilon}\log\frac{\log n}{\varepsilon}\right)$. To update κ_t and η_t^B, we extract κ_t from the list α_{γ_t} and $n_{c',t}$ from the list $\overline{\mathcal{C}}_{\gamma_t,t}$ in time $\mathcal{O}\left(\log|\overline{\mathcal{C}}_{\gamma_t}|\right) = \mathcal{O}\left(\frac{1}{\varepsilon}\log\frac{\log n}{\varepsilon}\right)$ by binary search. If the algorithm needs to update γ_t as well, this can be done by binary search on the list $\mathcal{G}_t$ in time $\mathcal{O}(\log|\mathcal{G}_t|) = \mathcal{O}\left(\log\frac{\log(n)}{\varepsilon}\right)$.

In both cases, the running time of answering the query and possibly updating data structures is bounded by the running time of the linear grouping step and by the routine to access one particular knapsack, i.e., by $\mathcal{O}\Big(\log m + \frac{\log n}{\varepsilon^2}\Big)$. □

Lemma 7.36. *The solution value can be calculated in time* $\mathcal{O}\Big(\frac{\log^3 n}{\varepsilon^4}\Big)$.

Proof. For obtaining the value of the current solution, we calculate the total value of the first $\bar{n}_t$ items. We do this by iterating through the value classes once and per value class, we iterate once through the list $\mathcal{T}_\ell$ to access the number $\bar{n}_t$. Then, we use prefix computation twice in order to access the total value of the first $\bar{n}_t$ items of type t. Lemma 7.2 bounds this time by $\mathcal{O}(\log n)$. By Lemma 7.9, the number of item types is bounded by $\mathcal{O}\Big(\frac{\log^2 n}{\varepsilon^4}\Big)$. Combining these two values bounds the total running time by $\mathcal{O}\Big(\frac{\log^3 n}{\varepsilon^4}\Big)$. As this time is clearly dominated by obtaining the implicit solution in the first place, we calculate and store the solution value when computing the implicit solution value and thus are able to return it in constant time. □

Lemma 7.37. *In time* $\mathcal{O}\Big(|P|\frac{\log^3 n}{\varepsilon^4}\Big(\log m + \frac{\log n}{\varepsilon^2}\Big)\Big)$ *a query for the complete solution* P *can be answered.*

Proof. For returning the complete solution, we determine the packed items and query each packed item individually. Lemma 7.35 bounds their query times by $\mathcal{O}\Big(\log m + \frac{\log n}{\varepsilon^2}\Big)$ while Lemma 7.2 bounds the running time for accessing item j. Lemma 7.9 bounds the number of item types by $\mathcal{O}\Big(\frac{\log^2 n}{\varepsilon^4}\Big)$. In total, the running time is bounded by $\mathcal{O}\Big(|P|\frac{\log^3 n}{\varepsilon^4}\Big(\log m + \frac{\log n}{\varepsilon^2}\Big)\Big)$, where P is the current solution. □

Proof of main result

Proof of Theorem 7.27. Lemma 7.31 gives the bound on the approximation ratio of our algorithm and Lemma 7.32 bounds the running time of an update operation. Further, Lemma 7.35 gives the running time for query operations. □

7.6 Special Knapsacks When Solving Multiple Knapsack

We give a high-level overview of our dynamic algorithm for a Multiple Knapsack instance with arbitrarily many knapsacks. While theoretically applicable for any number of knapsacks, the running time is reasonable when $m = \Big(\frac{1}{\varepsilon}\log n\Big)^{\mathcal{O}_\varepsilon(1)}$. For the technical details and the complete analysis we refer to [BEM+20] and the PhD thesis of L. Nölke. Let $\overline{v}$ be an upper bound on $v_{\max}$ known in advance. The main result of this section is the following theorem.

Theorem 7.38. *For* $\varepsilon > 0$, *there is a dynamic* $(1+\varepsilon)$*-approximate algorithm for* Multiple Knapsack *with update time* $2^{f(1/\varepsilon)}\Big(\frac{m}{\varepsilon}\log(nv_{\max})\Big)^{\mathcal{O}(1)} + \mathcal{O}\Big(\frac{1}{\varepsilon}\log\overline{v}\log n\Big)$, *with* f *quasi-linear. Moreover, item queries are answered in time* $\mathcal{O}\Big(\log\frac{m}{\varepsilon}\log n\Big)$, *solution value queries in time* $\mathcal{O}(1)$, *and queries of the entire solution* P *in time* $\mathcal{O}\Big(|P|\log\frac{m}{\varepsilon}\log n\Big)$.

7.6.1 Algorithm

Definitions and data structures Let Opt be the set of items used in an optimal solution and $\text{Opt}_{\frac{m}{\varepsilon^2}}$ the set containing the $\frac{m}{\varepsilon^2}$ most valuable items of Opt; in both cases, break ties in favor of smaller items.

When computing the solution, we will assign low-value items fractionally. To this end, consider an item j and let v be such that $0 \le v \le v_j$. Then, the *proportional size* of item j of value v is defined as $s_j\frac{v}{v_j}$.

To efficiently run our algorithm we maintain several data structures. We store the items of each non-empty value class V_ℓ (at most $\lceil\log_{1+\varepsilon} v_{\max}\rceil + 1$) in a data structure ordered by non-decreasing size. Second, for each possible value class V_ℓ (at most $\lceil\log_{1+\varepsilon}\overline{v}\rceil + 1$), we maintain a data structure ordered by non-increasing density that contains all items of value $(1+\varepsilon)^\ell$ or lower. In particular, we maintain such a data structure even if V_ℓ is empty since initialization

is prohibitively expensive in terms of running time. We constantly maintain all data structures leading to the additive term in the update time of $\mathcal{O}(\log n \log_{1+\varepsilon} \overline{v})$. We use additional data structures to store our solution and support queries.

Algorithm The approach itself can be divided into two parts that consider high- and low-value items, respectively. The corresponding partition is guessed such that the high-value items contain the $\frac{m}{\varepsilon^2}$ most valuable items of an optimal solution Opt and the low-value items the remaining items of Opt. For the important high-value items, a good solution is paramount, so we employ an EPTAS for Multiple Knapsack. It is run on a low-cardinality set of high-value candidate items together with $\frac{m}{\varepsilon}$ placeholders of equal size that reserve space for low-value items. The values of placeholders are determined by filling them fractionally with the densest low-value items.

More precisely, we start by guessing $\ell_{\max}$, the index of the highest value class with items in Opt. Let $\tilde{\ell}_{\min}$ denote the index of the lowest value class that we need to consider for the $\frac{m}{\varepsilon^2}$ most valuable items, that is, $\tilde{\ell}_{\min} = \ell_{\max} - \left\lceil \log_{1+\varepsilon} \frac{m}{\varepsilon^3} \right\rceil$. We consider each $\tilde{\ell} \in \{\tilde{\ell}_{\min}, \dots, \ell_{\max}\}$ as possible guess $\tilde{\ell}$ for the index of the lowest value class with items in $\textsc{Opt}_{\frac{m}{\varepsilon^2}}$. For this value class, we additionally guess $n_{\min}$, the number of items of value class $V_{\tilde{\ell}}$ belonging to $\textsc{Opt}_{\frac{m}{\varepsilon^2}}$. There are at most $\frac{m}{\varepsilon^2}$ guesses to consider. Given these three guesses, let $H_{\frac{m}{\varepsilon^2}}$ denote the set of *candidates* for the set $\textsc{Opt}_{\frac{m}{\varepsilon^2}}$. That is, $H_{\frac{m}{\varepsilon^2}}$ contains the $\frac{m}{\varepsilon^2}$ smallest items from each value class V_ℓ with $\ell \in \{\tilde{\ell}+1, \dots, \ell_{\max}\}$ and the $n_{\min}$ smallest items from value class $V_{\tilde{\ell}}$.

Now consider the data structure containing all the items of value at most $(1+\varepsilon)^{\tilde{\ell}}$ sorted by decreasing density. From this data structure, we (temporarily) remove the $n_{\min}$ smallest items of value class $V_{\tilde{\ell}}$. After having completed the calculation for the current set of guesses, we insert the removed items again.

Next, we guess v_L, the total value of low-value items in Opt. We use the just modified data structure to determine the size of the densest low-value items that have a total value v_L, by possibly cutting the last item. That is, we consider the set $\mathcal{J}'$ of densest items such that $\sum_{j\in\mathcal{J}'} v_j < v_L \leq \sum_{j\in\mathcal{J}'} v_j + v_{j^\star}$, where $j^\star$ is the densest item not in $\mathcal{J}'$. Next, we add a piece of item $j^\star$ of value $v_L - \sum_{j\in\mathcal{J}'} v_j$ and proportional size $s_{j^\star} \frac{v_L - \sum_{j\in\mathcal{J}'} v_j}{v_{j^\star}}$ to $\mathcal{J}'$. Given this "block" of items, we create bundles $B_1, \dots, B_{\frac{m}{\varepsilon}}$ of equal value $\frac{\varepsilon}{m} v_L$ by cutting the block at appropriate points. The size of bundle B_k is the total size of the items completely contained in B_k plus the proportional size of the at most two fractional items belonging to B_k.

Next, we consider an instance of Multiple Knapsack with m knapsacks, the items $H_{\frac{m}{\varepsilon^2}}$, and an item per placeholder bundle B_k with the size as detailed above and value $\frac{\varepsilon}{m} v_L$. On this instance, we run the EPTAS designed by Jansen [Jan12], parameterized by ε, to obtain a packing P.

Among all guesses, we take the solution P with the highest value and retrace the removal of high-value items to obtain the data structure corresponding to the solution. These items are inserted again right before the next update operation. For each knapsack, we place the

items in $H_{\frac{m}{\varepsilon^2}}$ as indicated by P and all low-value items completely contained in any bundle B_k that is packed by P in this knapsack. While used candidates (and their packing) can be stored explicitly, low-value items are given only implicitly by storing the correct guesses and recomputing B_k on a query. We summarize the algorithm in Algorithm 7.7.

Algorithm 7.7: Dynamic algorithm for special knapsacks

guess $\ell_{\max}$, $\tilde{\ell}$, and $n_{\min}$
  compute high-value candidates $H_{\frac{m}{\varepsilon^2}}$
  guess v_L
    create placeholder bundles of low-value items $B_1, \ldots, B_{\frac{m}{\varepsilon}}$
    run an EPTAS on $H_{\frac{m}{\varepsilon^2}}$ and $B_1, \ldots, B_{\frac{m}{\varepsilon}}$

Queries We briefly explain how to handle the different types of queries.

- **Single Item Query:** If the queried item is contained in $H_{\frac{m}{\varepsilon^2}}$, its packing is stored explicitly. For low-value items, we store the first and last item completely contained in a bundle. On query of an item, we decide its membership in a bundle by comparing its density with the pivot elements (breaking ties by index). The packing of the corresponding bundle is given again explicitly.

- **Solution Value Query:** After each update operation, we compute and store the solution value. To this end, we compute the total value of the packed items in $H_{\frac{m}{\varepsilon^2}}$. For low-value items, we compute the total value of the items completely contained in a bundle that is assigned to a knapsack. Prefix computation on the data structure of low-value items enables us to handle this efficiently. Summing the total value of the candidates and the values of those low-value items yields the value of the current solution.

- **Entire Solution Query:** For the packed candidates, we output their stored packing. For the low-value items, we iterate over the items in packed bundles in the density sorted data structure and skip all fractionally packed items and all bundles not packed by the current solution.

7.7 Solving Multiple Knapsack

Having laid the groundwork with the previous two sections, we finally show how to maintain $(1+\varepsilon)$-approximate solutions for arbitrary instances of the Multiple Knapsack problem and give the main result of this chapter.

Theorem 7.39. *For each $\varepsilon > 0$, there is a dynamic $(1+\varepsilon)$-approximate algorithm for* Multiple Knapsack *with update time $2^{f(1/\varepsilon)}\left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}(\log m \log S_{max} \log v_{\max})^{\mathcal{O}(1)}$, where $f\left(\frac{1}{\varepsilon}\right)$ is*

quasi-linear. Item queries can be answered in time $\mathcal{O}(\log m)\left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1)}$ *and the solution* P *can be output in time* $\mathcal{O}(|P| + \log m)\left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1)}$.

We obtain this result by partitioning the knapsacks into three sets, special, extra, and ordinary knapsacks, and solving the respective subproblems. This has similarities to the approach in [Jan09]; however, there it is sufficient to have only two groups of knapsacks. In Section 7.5 we develop the algorithmic techniques used for solving the ordinary subproblem, and Section 7.6 gives a high-level overview of the algorithm for the special subproblem.

7.7.1 Algorithm

Definitions and data structures Given n and ε, let L denote the number of knapsacks needed as resource augmentation by the algorithm for the ordinary subproblem (Section 7.5). We can choose $L \in \left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}$. Further, we assume that $m > \left(\frac{1}{\varepsilon}\right)^{4/\varepsilon} \cdot L$ because otherwise the algorithm for the special subproblem (Section 7.6) has update time polynomial in $\log n$. Let Opt denote the set of items in an optimal solution; we break ties by picking smaller-size items.

We partition the knapsacks into three parts that decrease in knapsack capacity but increase in cardinality. We refer to them as *special*, *extra*, and *ordinary* knapsacks, with special denoting the largest knapsacks, ordinary the smallest, and extra the in-between ones. We call an item *ordinary* if it fits into the at least one ordinary knapsack and *special* otherwise. We denote the set of ordinary and special items by $\mathcal{J}_O$ and by $\mathcal{J}_S$, respectively.

Similar to the proportional size, we define the *proportional value*. For an item j with size s_j and value v_j and a given size $s \leq s_j$, the proportional value is given by $v_j \frac{s}{s_j}$.

Since we use the algorithms from Sections 7.5 and 7.6 as subroutines, we require the maintenance of the corresponding data structures. However, the data structures containing all items of value class at most V_ℓ will be set up on the fly. Hence, we do not maintain them with the update operations. That is, we only maintain one data structure storing all items sorted by index, one data structure for all knapsacks sorted by non-increasing capacity, and per value class we store all its items sorted by non-decreasing size.

Algorithm The high-level idea of the algorithm is the following. We start by partitioning the knapsacks into special, extra, and ordinary knapsacks; this partition depends on the current instance and is computed in each update operation. The extra knapsacks provide additional knapsacks needed when solving the special and the ordinary subproblems. Since ordinary items may be packed in special and ordinary knapsacks, we also guess s_O, the *size* of ordinary items packed in special knapsacks up to powers of $(1 + \varepsilon)$. Next, we add a virtual knapsack of size s_O to the ordinary subproblem and solve the resulting instance with the algorithm for ordinary knapsacks. Here, our choice for the cardinality of the extra knapsacks will enable us to treat these as knapsacks given by resource augmentation. Items that are packed in the virtual

knapsack are then (possibly fractionally) assigned to bundles. These bundles together with the special items constitute the input to the special subproblem solved with the corresponding algorithm.

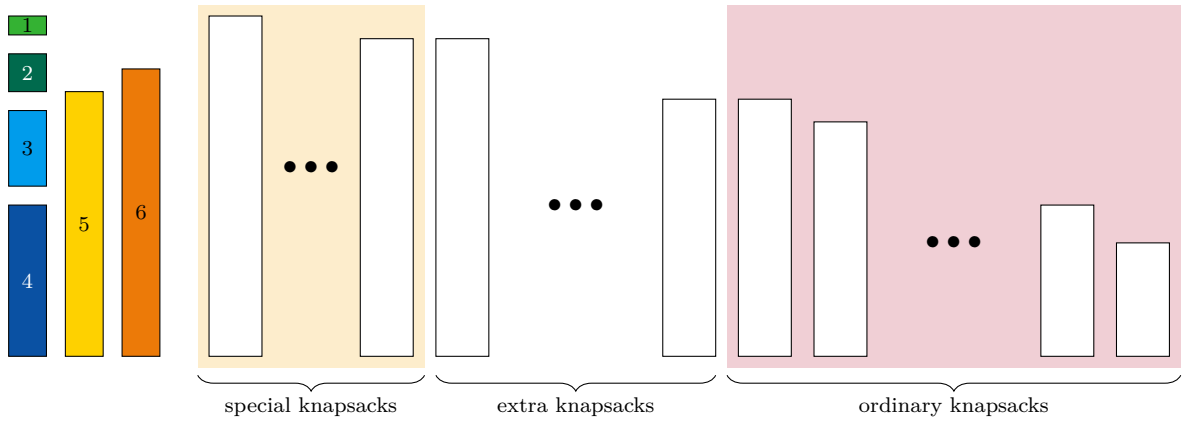

Figure 7.8: Special, extra, and ordinary knapsacks with special (5 and 6) and ordinary (1 through 4) items

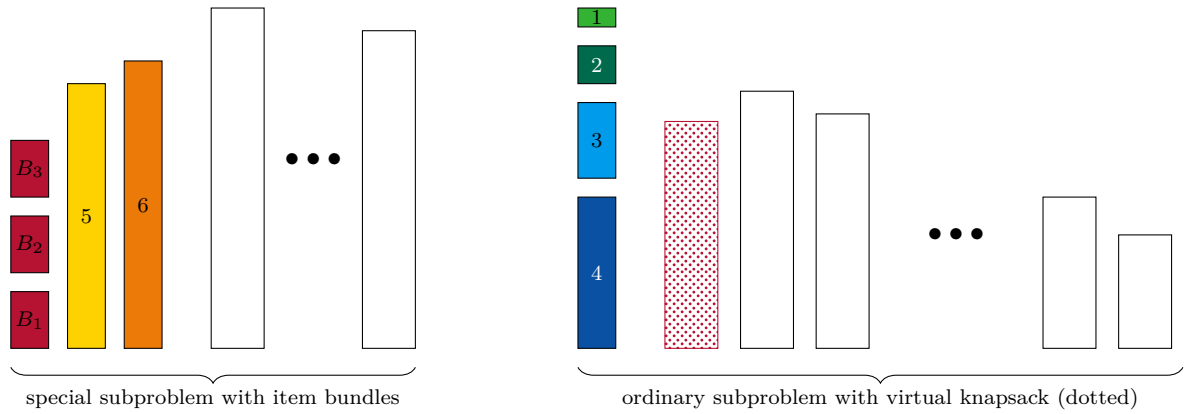

Figure 7.9: Special and ordinary subproblem

More precisely, we group the $L\sum_{i=1}^{1/\varepsilon}\frac{1}{\varepsilon^{3(i-1)}}$ largest knapsacks of the instance in $\frac{1}{\varepsilon}$ sets of geometrically increasing cardinality and non-increasing capacity. The first set contains the L largest knapsacks and, in general, for $i \in \left[\frac{1}{\varepsilon}\right]$, the ith set contains the $\frac{L}{\varepsilon^{3(i-1)}}$ largest knapsacks not yet contained in a set with smaller index. Then, we guess the index k of the *last* such group that still contains special knapsacks. The next $\frac{L}{\varepsilon^{3k}}$ knapsacks are the extra knapsacks, and all remaining knapsacks constitute the set of ordinary knapsacks.

Next, we apply dynamic linear grouping to transform the items into item types. Let L_S denote the number of special knapsacks, i.e., $L_S = \sum_{i=1}^{k}\frac{L}{\varepsilon^{3(i-1)}}$. We pack each of the $\frac{L_S}{\varepsilon^2}$ most valuable ordinary items, denoted by $\mathcal{J}_E$, in a separate extra knapsack, storing their explicit packing and removing them temporarily from the data structure of their respective value class.

The remaining ordinary items are now considered as input to the ordinary subproblem. Further, we guess s_O, the size of ordinary items packed in special knapsacks to create a virtual ordinary knapsack of capacity s_O. As guessing s_O exactly is intractable, we only guess s_O up to a factor of $(1+\varepsilon)$. Next, we run the algorithm for ordinary knapsacks as specified in Section 7.5. When doing this, we treat the virtual knapsack as its own group and do not create configurations for this knapsack but restrict to the z variables, that place items of a particular type by number. When rounding the variables (also for the virtual knapsack), we use the extra knapsacks for providing resource augmentation. This gives a packing of ordinary items either in ordinary knapsacks, in the virtual knapsack, or in the extra knapsacks.

Let $\bar{s}_O$ denote the total size of ordinary items in the virtual knapsack. We sort the items placed inside by type, i.e., first by value, then by size, and cut the virtual knapsack to create $\frac{L_S}{\varepsilon}$ bundles of equal size. This may lead to some ordinary items being contained fractionally in more than one bundle. Such items will not be packed in the final solution but their proportional value contributes to the value of a bundle. We denote by $\mathcal{B}_O$ the set of bundles, and, for each bundle, we remember how many items of each type are completely contained. Then, each bundle B is considered as one item of value equal to the proportional value of the items placed in B; the size of B is $\frac{\varepsilon \bar{s}_O}{L_S}$.

Next, we set up the data structures used in the special subproblem containing only special items (as types) and the set of bundles $\mathcal{B}_O$. That is, for each value class V_ℓ we create one data structure that contains only the items of this value class sorted by increasing size and one data structure that contains all items of at most this value class sorted by density. Note that the values of the ordinary bundles are not necessarily rounded to powers of $(1+\varepsilon)$. However, since their number is bounded, we (possibly) create a value class for each new value and treat them as we would treat the regular value classes. Note that we do not insert every special item but only special item types and their multiplicities. Having set up these data structures, we run the algorithm for special knapsacks as described in Section 7.6.

Algorithm 7.8: Dynamic algorithm for Multiple Knapsack

```
guess k
    partition knapsacks into special, extra, and ordinary knapsacks
    guess ℓ_max and use dynamic linear grouping
        pack the L_S/ε² most valuable ordinary items into extra knapsacks
        guess s_O and create a virtual ordinary knapsack of size s_O
            solve the ordinary subproblem including the virtual knapsack
            create bundles of ordinary items
            set up the data structure for the special subproblem considering only special items and
                these bundles
            solve the special subproblem
```

Queries For handling queries, we essentially use the same approach as in Sections 7.5 and 7.6 for the ordinary and special subproblem, respectively. By default, we pack the first $\bar{n}_t$ items of a type that contributes $\bar{n}_t$ items to the implicit solution. However, we point out two steps that change the routines slightly. First, we incorporate handling item types as described in Section 7.5 by setting up pointers also for the special items assigned to special knapsacks. Second, extra care has to be taken for the items in the virtual knapsack. To this end, we store the number of items per type that are completely contained in bundles that are chosen by the special subproblem and only add the number of such items to the overall number of packed items of a certain type. Further, we additionally maintain a pointer for each type that points to the bundle in the virtual knapsack where the next item of this type is assigned to. If the query for an ordinary item returns the virtual knapsack, we use the corresponding pointer to determine the bundle of this item. Then, we query the *bundle* as item in the special subproblem and return the answer for this item.

7.7.2 Analysis

In this section, we analyze the performance of our algorithm in terms of the solution quality and in terms of the update time. We heavily rely on the results of the previous sections that guarantee that the solutions to our subproblems are sufficiently good.

Lemma 7.40. *Let P_F be the final solution the algorithm computes and let* Opt *be an optimal solution. We have $v(P_F) \geq \frac{(1-9\varepsilon)(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^4} v(\text{Opt})$.*

Proof. Our algorithm returns the solution P_F that has the highest total value among all solutions found when trying guesses. If we can construct another solution P that our algorithm might have considered at some point, then the lemma follows since $v(P_F) \geq v(P)$. To this end, let Opt be an optimal solution and let $\ell_{\max}$ be the largest index of a value class with $V_\ell \cap \text{Opt} \neq \emptyset$. We fix an optimal solution $\text{Opt}_\mathcal{T}$ of the instance induced by the set of item types $\mathcal{T}$, obtained when running dynamic linear grouping with $\ell_{\max}$, and their multiplicities.

Further, we consider the $\frac{1}{\varepsilon}$ many groups of the largest knapsacks that we used to partition the knapsacks into special, extra, and ordinary. As there are $\frac{1}{\varepsilon}$ many such groups, one of these groups contributes at most $\varepsilon v(\text{Opt}_\mathcal{T})$ to the total solution value. Let $k \in \left[\frac{1}{\varepsilon}\right]$ be such that the $(k+1)$st group is such a low-value group. This value k gives us the partition for the knapsacks: the first $L \sum_{i=1}^{k} \frac{1}{\varepsilon^{3(i-1)}}$ many knapsacks are the special ones, the knapsacks in group $k+1$ are the extra knapsacks, and all remaining knapsacks are ordinary. Based on this partition, we also group the items into ordinary and special items.

Let $\tilde{s}_O$ be the total size of ordinary items packed by $\text{Opt}_\mathcal{T}$ in special knapsacks. Thus, our algorithm considered the combination of $\ell_{\max}$, k, and $s_O = (1+\varepsilon)^{\lfloor \log_{1+\varepsilon} \tilde{s}_O \rfloor}$ at some point by definition. The solution P is obtained by modifying $\text{Opt}_\mathcal{T}$ in a way that allows P to be a possible solution for the algorithm when considering these three guesses.

More precisely, we start by removing the items from the extra knapsacks to obtain a solution $\textsc{Opt}'_{\mathcal{T}}$. By our choice of k, we have $v(\textsc{Opt}'_{\mathcal{T}}) \geq (1-\varepsilon)v(\textsc{Opt}_{\mathcal{T}})$. Let $\textsc{Opt}_{-E}$ denote an optimal solution of the instance consisting of all items (as types) and the ordinary as well as the special knapsacks. Thus,

$$v(\textsc{Opt}_{-E}) \geq (1-\varepsilon)v(\textsc{Opt}_{\mathcal{T}}).$$

Now, we consider the ordinary subproblem, i.e., the instance of Multiple Knapsack consisting of all ordinary knapsacks plus the virtual knapsack and all ordinary items except $\mathcal{J}_E$, the high-value items our algorithm packs in extra knapsacks. Take the packing of items in ordinary knapsacks as done by $\textsc{Opt}_{-E}$ and greedily, sorted by non-increasing density, pack the items that are packed in special knapsacks by $\textsc{Opt}_{-E}$ in the virtual knapsack without violating its capacity. As s_O underestimates the size of these items, this causes a loss of at most $\varepsilon v(\textsc{Opt}_{\mathcal{T}})$ plus an ordinary item j_O with $v_{j_O} \leq \min_{j\in\mathcal{J}_E} v_j \leq \varepsilon v(\textsc{Opt}_{\mathcal{T}})$. This packing is feasible for the ordinary subproblem as described above and achieves a value of at least $v((\textsc{Opt}_{-E} \cap \mathcal{J}_O) \setminus \mathcal{J}_E) - 2\varepsilon v(\textsc{Opt}_{\mathcal{T}})$. For an optimal solution $\textsc{Opt}_O$ to this instance,

$$v(\textsc{Opt}_O) \geq v(\textsc{Opt}_{-E,O}) - 2\varepsilon v(\textsc{Opt}_{\mathcal{T}}),$$

where $\textsc{Opt}_{-E,O} = (\textsc{Opt}_{-E} \cap \mathcal{J}_O) \setminus \mathcal{J}_E$.

We note that $\frac{L}{\varepsilon^{3k}} > \frac{L_S}{\varepsilon^2} + 2L$. Hence, the extra knapsacks can indeed act as resource augmentation for the ordinary subproblem. Let P_O denote the packing returned by the algorithm described in Section 7.5. By Theorem 7.27,

$$v(P_O) \geq \frac{1}{1+\varepsilon} v(\textsc{Opt}_O).$$

With $\textsc{Opt}_{-E,S} := \textsc{Opt}_{-E} \cap \mathcal{J}_S$ and $\textsc{Opt}_{-E,E} := \textsc{Opt}_{-E} \cap \mathcal{J}_E$, we can rewrite $\textsc{Opt}_{-E}$ as $\textsc{Opt}_{-E,O} \cup \textsc{Opt}_{-E,E} \cup \textsc{Opt}_{-E,S}$. Let $P_1 = P_O \cup \textsc{Opt}_{-E,S} \cup \mathcal{J}_E$. Thus,

$$\begin{aligned}
v(P_1) &= v(P_O) + v(\textsc{Opt}_{-E,S}) + v(\mathcal{J}_E) \\
&\geq \frac{1}{1+\varepsilon} v(\textsc{Opt}_O) + v(\textsc{Opt}_{-E,S}) + v(\textsc{Opt}_{-E,E}) \\
&\geq \frac{1}{1+\varepsilon} v(\textsc{Opt}_{-E,O}) - 2\varepsilon v(\textsc{Opt}_{\mathcal{T}}) + v(\textsc{Opt}_{-E,S}) + v(\textsc{Opt}_{-E,E}) \\
&\geq \frac{1}{1+\varepsilon} v(\textsc{Opt}_{-E}) - 2\varepsilon v(\textsc{Opt}_{\mathcal{T}}) \\
&\geq \left(\frac{1-\varepsilon}{1+\varepsilon} - 2\varepsilon\right) v(\textsc{Opt}_{\mathcal{T}}).
\end{aligned}$$

Now, we use P_1 to obtain the packing P that our algorithm could have considered at some point. To this end, we observe that P_1 still uses the virtual knapsack while our algorithm packs ordinary items via bundles in special knapsacks. Thus, we take the items in the virtual

knapsack in P_1 and transform them into $\frac{L_S}{\varepsilon}$ equal-size bundles (with possibly cut items) to obtain the intermediate packing P_2. The packing of $\text{OPT}_{-E,S}$ reserves sufficient space to pack these bundles fractionally into the special knapsacks. If we arrange them such that the at most L_S fractionally packed bundles are those that have lowest value, we can discard these at a cost of at most $\varepsilon v(\text{OPT}_{\mathcal{T}})$. Further, we remove the ordinary items that are now part of more than one bundle. Since there are at most $\frac{L_S}{\varepsilon}$ such items and we packed the $\frac{L_S}{\varepsilon^2}$ most valuable ordinary items in extra knapsacks, the cost of this removal is bounded by $\varepsilon v(\text{OPT}_{\mathcal{T}})$. Hence,

$$v(P_2) \geq v(P_1) - 2\varepsilon v(\text{OPT}_{\mathcal{T}}) \geq \left(\frac{1-\varepsilon}{1+\varepsilon} - 4\varepsilon\right) v(\text{OPT}_{\mathcal{T}}) \geq \frac{1-9\varepsilon}{1+\varepsilon} v(\text{OPT}_{\mathcal{T}}).$$

Further, combining the bundles of ordinary items and the special items creates a valid solution to the special subproblem as solved in Section 7.6. Hence, the solution returned by the algorithm on these particular guesses $\ell_{\max}$, k, and s_O satisfies $v(P) \geq \frac{1}{1+\varepsilon} v(P_2)$ by Theorem 7.38. Combining the calculations and using Theorem 7.4, we conclude

$$v(P_F) \geq v(P) \geq \frac{1}{1+\varepsilon} v(P_2) \geq \frac{1-9\varepsilon}{(1+\varepsilon)^2} v(\text{OPT}_{\mathcal{T}}) \geq \frac{(1-9\varepsilon)(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^4} v(\text{OPT}).$$

□

Next, we bound the update time of our algorithm.

Lemma 7.41. *The algorithm has update time* $2^{f(1/\varepsilon)} \left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)} (\log m \log S_{max} \log v_{\max})^{\mathcal{O}(1)}$, *where* f *is a quasi-linear function.*

Proof. Our algorithm heavily relies on solving the subproblems efficiently. The running times of the algorithms are given in Theorems 7.27 and 7.38. However, we note that we do not maintain the data structures for the special subproblem but create these on the fly after having determined the item types. Hence, we do not have the additive term for maintaining the at most $\mathcal{O}(\log_{1+\varepsilon} \overline{v})$ data structures of items with value up to $(1+\varepsilon)^\ell$ for $\ell \in \{0, \ldots, \lfloor \log_{1+\varepsilon} \overline{v} \rfloor\}$ sorted by non-increasing density.

Further, guessing k adds a factor of $\frac{1}{\varepsilon}$ to the update time. Placing the $\frac{L_S}{\varepsilon^2}$ most valuable ordinary items in extra knapsacks and removing them from data structures takes time $\mathcal{O}\left(\frac{L_S}{\varepsilon^2} \log n\right)$ which is within the time bound. The same holds for updating the ordinary data structures and generating the special data structures as well as solving the respective subproblems.

It remains to show that the bundles of ordinary items stemming from the virtual knapsack can be generated efficiently. To this end, we sort the item types in some fixed order and store the number of items per type that are packed in the knapsack. There are at most $\left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1)}$ item types and at most $\left(\frac{\log n}{\varepsilon}\right)^{\mathcal{O}(1/\varepsilon)}$ many bundles. Hence, we can iterate through the list of item types and obtain the cutting points using prefix computation as explained in Section 7.2 within the desired time frame. □

Answering queries As explained above, the queries are handled as described in the respective subproblems with the exception that the special subproblem also deals with item types and that ordinary items packed in the virtual knapsack lead to an additional query of their bundle in the special subproblem. Therefore, the bounds for answering queries given in the respective section immediately imply the following lemma.

Lemma 7.42. *The query times of our algorithm are as follows.*

(i) Single item queries are answered in time $\mathcal{O}\Big(\log m + \frac{\log n}{\varepsilon^2} + \frac{\log(\log n/\varepsilon)}{\varepsilon}\Big)$.

(ii) Solution value queries are answered in time $\mathcal{O}(1)$.

(iii) The entire solution P *can be output in time* $\mathcal{O}\Big(|P|\frac{\log^3 n}{\varepsilon^4}\Big(\log m + \frac{\log n}{\varepsilon^2} + \frac{\log(\log n/\varepsilon)}{\varepsilon}\Big)\Big)$.

Proof of main result We are now ready to prove Theorem 7.39.

Proof of Theorem 7.39. Lemma 7.40 bounds the quality of the solution found by our dynamic algorithm. Lemma 7.41 and Lemma 7.42 give the bounds on the running times of update and query operations, respectively. □

7.8 Concluding Remarks

We have presented a robust dynamic framework for Multiple Knapsack that implements update operations (item and knapsack arrivals and departures) as well as query operations, such as solution value and item presence in the solution. By having n items arrive one by one, any dynamic algorithm can be turned into a non-dynamic framework while incurring an additional linear term in the running time. Hence, the performance of any dynamic framework is subject to the same lower bounds as non-dynamic approximation schemes.

We hope to foster further research within the dynamic-algorithm framework for packing, scheduling and, generally, non-graph problems. For bin packing and for scheduling to minimize the makespan on uniformly related machines, we note that existing PTAS techniques from [KK82] and [Jan10, HS87] combined with rather straightforward data structures can be lifted to a fully dynamic framework for the respective problems.

References

[AAG+19] A. Abboud, R. Addanki, F. Grandoni, D. Panigrahi, and B. Saha. Dynamic set cover: Improved algorithms and lower bounds. In *STOC*, pages 114–125. ACM, 2019. doi:10.1145/3313276.3316376.

[AAPW01] B. Awerbuch, Y. Azar, S. A. Plotkin, and O. Waarts. Competitive routing of virtual circuits with unknown duration. *J. Comput. Syst. Sci.*, 62(3):385–397, 2001. doi:10.1006/jcss.1999.1662.

[ABK94] Y. Azar, A. Z. Broder, and A. R. Karlin. On-line load balancing. *Theor. Comput. Sci.*, 130(1):73–84, 1994. doi:10.1016/0304-3975(94)90153-8.

[AF55] S. B. Akers and J. Friedman. A non-numerical approach to production scheduling problems. *Journal of the Operations Research Society of America*, 3(4):429–442, 1955. doi:10.1287/opre.3.4.429.

[AGZ99] M. Andrews, M. X. Goemans, and L. Zhang. Improved bounds for on-line load balancing. *Algorithmica*, 23(4):278–301, 1999. doi:10.1007/PL00009263.

[AKL+15] Y. Azar, I. Kalp-Shaltiel, B. Lucier, I. Menache, J. Naor, and J. Yaniv. Truthful online scheduling with commitments. In *EC*, pages 715–732. ACM, 2015. doi:10.1145/2764468.2764535.

[AKP+97] Y. Azar, B. Kalyanasundaram, S. A. Plotkin, K. Pruhs, and O. Waarts. On-line load balancing of temporary tasks. *J. Algorithms*, 22(1):93–110, 1997. doi:10.1006/jagm.1995.0799.

[Alb99] S. Albers. Better bounds for online scheduling. *SIAM J. Comput.*, 29(2):459–473, 1999. doi:10.1137/S0097539797324874.

[Alb03] S. Albers. Online algorithms: a survey. *Math. Program.*, 97(1-2):3–26, 2003. doi:10.1007/s10107-003-0436-0.

[ALLM18] K. Agrawal, J. Li, K. Lu, and B. Moseley. Scheduling parallelizable jobs online to maximize throughput. In *LATIN*, volume 10807, pages 755–776. Springer, 2018. doi:10.1007/978-3-319-77404-6_55.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows*. Prentice Hall, 1993. https://www.pearson.com/us/higher-education/program/Ahuja-Network-Flows-Theory-Algorithms-and-Applications/PGM148966.html.

[ANR92] Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignments. In *SODA*, pages 203–210. ACM/SIAM, 1992. http://dl.acm.org/citation.cfm?id=139404.139450.

[AW14] A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.53.

[Bay72] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972. doi:10.1007/BF00289509.

[BB10] A. Beloglazov and R. Buyya. Energy efficient allocation of virtual machines in cloud data centers. In *CCGRID*, pages 577–578. IEEE Computer Society, 2010. doi:10.1109/CCGRID.2010.45.

[BC16] F. Biagini and M. Campanino. *Elements of Probability and Statistics*. Springer, 1st edition, 2016. doi:10.1007/978-3-319-07254-8.

[BCP11] N. Bansal, H. Chan, and K. Pruhs. Competitive algorithms for due date scheduling. *Algorithmica*, 59(4):569–582, 2011. doi:10.1007/s00453-009-9321-4.

[BDF81] J. L. Bruno, P. J. Downey, and G. N. Frederickson. Sequencing tasks with exponential service times to minimize the expected flow time or makespan. *J. ACM*, 28(1):100–113, 1981. doi:10.1145/322234.322242.

[BE98] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998. https://www.cambridge.org/de/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/online-computation-and-competitive-analysis?format=PB.

[Bel56] R. Bellman. Mathematical aspects of scheduling theory. *J. Soc. Indust. Appl. Math.*, 4(3):168–205, 1956. doi:10.1137/0104010.

[Bel57] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957. https://press.princeton.edu/books/paperback/9780691146683/dynamic-programming.

[BEM+20] M. Böhm, F. Eberle, N. Megow, L. Nölke, J. Schlöter, B. Simon, and A. Wiese. Fully dynamic algorithms for knapsack problems with polylogarithmic update time. *CoRR*, abs/2007.08415, 2020. https://arxiv.org/abs/2007.08415.

[BH97] S. K. Baruah and J. R. Haritsa. Scheduling for overload in real-time systems. *IEEE Trans. Computers*, 46(9):1034–1039, 1997. doi:10.1109/12.620484.

[BHI15] S. Bhattacharya, M. Henzinger, and G. F. Italiano. Design of dynamic algorithms via primal-dual method. In *ICALP (1)*, volume 9134 of *Lecture Notes in Computer Science*, pages 206–218. Springer, 2015. doi:10.1007/978-3-662-47672-7_17.

[BHN17] S. Bhattacharya, M. Henzinger, and D. Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in $O(\log^3 n)$ worst case update time. In *SODA*, pages 470–489. SIAM, 2017. doi:10.1137/1.9781611974782.30.

[BHN19] S. Bhattacharya, M. Henzinger, and D. Nanongkai. A new deterministic algorithm for dynamic set cover. In *FOCS*, pages 406–423. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00033.

[BHR19] A. Bernstein, J. Holm, and E. Rotenberg. Online bipartite matching with amortized $O(\log^2 n)$ replacements. *J. ACM*, 66(5):37:1–37:23, 2019. doi:10.4230/LIPIcs.ITCS.2017.51.

[BHS94] S. K. Baruah, J. R. Haritsa, and N. Sharma. On-line scheduling to maximize task completions. In *RTSS*, pages 228–236. IEEE Computer Society, 1994. doi:10.1109/REAL.1994.342713.

[BJK20] S. Berndt, K. Jansen, and K. Klein. Fully dynamic bin packing revisited. *Math. Program.*, 179(1):109–155, 2020. doi:10.1007/s10107-018-1325-x.

[BK19] S. Bhattacharya and J. Kulkarni. Deterministically maintaining a $(2 + \varepsilon)$-approximate minimum vertex cover in $o(1/\varepsilon^2)$ amortized update time. In *SODA*, pages 1872–1885. SIAM, 2019. doi:10.1137/1.9781611975482.113.

[BKB07] N. Bobroff, A. Kochut, and K. A. Beaty. Dynamic placement of virtual machines for managing SLA violations. In *Integrated Network Management*, pages 119–128. IEEE, 2007. doi:10.1109/INM.2007.374776.

[BKM+91] S. K. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. E. Rosier, and D. E. Shasha. On-line scheduling in the presence of overload. In *FOCS*, pages 100–110. IEEE Computer Society, 1991. doi:10.1109/SFCS.1991.185354.

[BKM+92] S. K. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. E. Rosier, D. E. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. *Real-Time Systems*, 4(2):125–144, 1992. doi:10.1007/BF00365406.

[BKP+17] A. Bernstein, T. Kopelowitz, S. Pettie, E. Porat, and C. Stein. Simultaneously load balancing for every p-norm, with reassignments. In *ITCS*, volume 67 of *LIPIcs*, pages 51:1–51:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ITCS.2017.51.

[BM72] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi:10.1007/BF00288683.

[BP11] N. Boria and V. T. Paschos. A survey on combinatorial optimization in dynamic environments. *RAIRO - Operations Research*, 45(3):241–294, 2011. doi:10.1051/ro/2011114.

[BRVW20] M. Buchem, L. Rohwedder, T. Vredeveld, and A. Wiese. Additive approximation schemes for load balancing problems. *CoRR*, abs/2007.09333, 2020. https://arxiv.org/abs/2007.09333.

[BT97] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*, volume 6 of *Athena Scientific Optimization and Computation Series*. Athena Scientific, 1997. http://athenasc.com/linoptbook.html.

[CEM+20] L. Chen, F. Eberle, N. Megow, K. Schewior, and C. Stein. A general framework for handling commitment in online throughput maximization. *Math. Prog.*, 183:215–247, 2020. doi:10.1007/s10107-020-01469-2.

[CFMM14] M. Cheung, F. Fischer, J. Matuschke, and N. Megow. An $\Omega(\Delta^{1/2})$ Gap example on the (W)SEPT Policy. Unpublished note, 2014.

[Cha18] T. M. Chan. Approximation schemes for 0-1 knapsack. In *SOSA@SODA*, volume 61 of *OASICS*, pages 5:1–5:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/OASIcs.SOSA.2018.5.

[CK05] C. Chekuri and S. Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM J. Comput.*, 35(3):713–728, 2005. doi:10.1137/S0097539700382820.

[CKPT17] H. I. Christensen, A. Khan, S. Pokutta, and P. Tetali. Approximation and online algorithms for multidimensional bin packing: A survey. *Comput. Sci. Rev.*, 24:63–79, 2017. doi:10.1016/j.cosrev.2016.12.001.

[CMM67] R. Conway, W. Maxwell, and L. Miller. *Theory of Scheduling*. Adison-Wesley Pub. Co., 1967.

[CMWW19] M. Cygan, M. Mucha, K. Wegrzycki, and M. Wlodarczyk. On problems equivalent to (min, +)-convolution. *ACM Trans. Algorithms*, 15(1):14:1–14:25, 2019. doi:10.1145/3293465.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971. doi:10.1145/800157.805047.

[DEGI10] C. Demetrescu, D. Eppstein, Z. Galil, and G. F. Italiano. *Dynamic Graph Algorithms*, pages 9:1–9:28. Chapman & Hall/CRC, 2 edition, 2010. doi:10.1201/9781584888239.

[DGV08] B. C. Dean, M. X. Goemans, and J. Vondrák. Approximating the stochastic knapsack problem: The benefit of adaptivity. *Math. Oper. Res.*, 33(4):945–964, 2008. doi:10.1287/moor.1080.0330.

[dlVL81] W. F. de la Vega and G. S. Lueker. Bin packing can be solved within 1+epsilon in linear time. *Combinatorica*, 1(4):349–355, 1981. doi:10.1007/BF02579456.

[DP00] B. DasGupta and M. A. Palis. Online real-time preemptive scheduling of jobs with deadlines. In *APPROX*, volume 1913 of *Lecture Notes in Computer Science*, pages 96–107. Springer, 2000. doi:10.1007/3-540-44436-X_11.

[DP09] D. Dubhashi and A. Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 1 edition, 2009. doi:10.1017/CBO9780511581274.

[EEI64] W. L. Eastman, S. Even, and I. M. Isaacs. Bounds for the optimal scheduling of n jobs on m processors. *Management Science*, 11(2):268–279, 1964. doi:10.1287/mnsc.11.2.268.

[EFMM19] F. Eberle, F. Fischer, J. Matuschke, and N. Megow. On index policies for stochastic minsum scheduling. *Oper. Res. Lett.*, 47(3):213–218, 2019. doi:10.1016/j.orl.2019.03.007.

[EL09] L. Epstein and A. Levin. A robust APTAS for the classical bin packing problem. *Math. Program.*, 119(1):33–49, 2009. doi:10.1007/s10107-007-0200-y.

[EL13] L. Epstein and A. Levin. Robust approximation schemes for cube packing. *SIAM J. Optim.*, 23(2):1310–1343, 2013. doi:10.1137/11082782X.

[EL14] L. Epstein and A. Levin. Robust algorithms for preemptive scheduling. *Algorithmica*, 69(1):26–57, 2014. doi:10.1007/s00453-012-9718-3.

[EMS20] F. Eberle, N. Megow, and K. Schewior. Optimally handling commitment issues in online throughput maximization. In *ESA*, volume 173 (to appear) of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[FBK+12] A. D. Ferguson, P. Bodík, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *EuroSys*, pages 99–112. ACM, 2012. doi:10.1145/2168836.2168847.

[FFG+18] B. Feldkord, M. Feldotto, A. Gupta, G. Guruganesh, A. Kumar, S. Riechers, and D. Wajc. Fully-dynamic bin packing with little repacking. In *ICALP*, volume 107 of *LIPIcs*, pages 51:1–51:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ICALP.2018.51.

[FW98] A. Fiat and G. J. Woeginger, editors. *Online Algorithms, The State of the Art (the book grow out of a Dagstuhl Seminar, June 1996)*, volume 1442 of *Lecture Notes in Computer Science*. Springer, 1998. doi:10.1007/BFb0029561.

[GG61] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Oper. Res.*, 9(6):849–859, 1961. doi:10.1287/opre.11.6.863.

[GGK16] A. Gu, A. Gupta, and A. Kumar. The power of deferral: Maintaining a constant-competitive steiner tree online. *SIAM J. Comput.*, 45(1):1–28, 2016. doi:10.1137/140955276.

[GGP97] L. Georgiadis, R. Guérin, and A. K. Parekh. Optimal multiplexing on a single link: Delay and buffer requirements. *IEEE Trans. Inf. Theory*, 43(5):1518–1535, 1997. doi:10.1109/18.623149.

[GGW11] J. C. Gittins, K. D. Glazebrook, and R. R. Weber. *Multi-Armed Bandit Allocation Indices*. John Wiley & Sons, Ltd, 2nd edition, 2011. doi:10.1002/9780470980033.

[GHKM14] K. D. Glazebrook, D. J. Hodge, C. Kirkbride, and R. J. Minty. Stochastic scheduling: A short history of index policies and new approaches to index generation for dynamic resource allocation. *J. Sched.*, 17(5):407–425, 2014. doi:10.1007/s10951-013-0325-1.

[GI99] A. Goel and P. Indyk. Stochastic load balancing and related problems. In *FOCS*, pages 579–586. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814632.

[Git79] J. C. Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society, Series B*, 41:148–177, 1979. doi:10.1111/j.2517-6161.1979.tb01068.x.

[Git89] J. C. Gittins. *Multi-Armed Bandit Allocation Indices*. Wiley, 1989.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GJL98] M. M. Güntzer, D. Jungnickel, and M. Leclerc. Efficient algorithms for the clearing of interbank payments. *Eur. J. Oper. Res.*, 106(1):212–219, 1998. doi:10.1016/S0377-2217(97)00265-8.

[GK03] M. H. Goldwasser and B. Kerbikov. Admission control with immediate notification. *J. Sched.*, 6(3):269–285, 2003. doi:10.1023/A:1022956425198.

[GKKP17] A. Gupta, R. Krishnaswamy, A. Kumar, and D. Panigrahi. Online and dynamic algorithms for set cover. In *STOC*, pages 537–550. ACM, 2017. doi:10.1145/3055399.3055493.

[GKNS18] A. Gupta, A. Kumar, V. Nagarajan, and X. Shen. Stochastic load balancing on unrelated machines. In *SODA*, pages 1274–1285. SIAM, 2018. doi:10.1137/1.9781611975031.83.

[GKS14] A. Gupta, A. Kumar, and C. Stein. Maintaining assignments online: Matching, scheduling, and flows. In *SODA*, pages 468–479. SIAM, 2014. doi:10.1137/1.9781611973402.35.

[GL79] G. Gens and E. Levner. Computational complexity of approximation algorithms for combinatorial problems. In *MFCS*, volume 74 of *Lecture Notes in Computer Science*, pages 292–300. Springer, 1979. doi:10.1007/3-540-09526-8_26.

[GL80] G. Gens and E. Levner. Fast approximation algorithms for knapsack type problems. In *Optimization Techniques*, pages 185–194. Springer, 1980. doi:10.1007/BFb0006603.

[Gla79] K. Glazebrook. Scheduling tasks with exponential service times on parallel processors. *J. Appl. Probab.*, 16(3):65–689, 1979. doi:10.2307/3213099.

[GLLRK79a] R. Graham, E. Lawler, J. Lenstra, and A. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979. doi:10.1016/S0167-5060(08)70356-X.

[GLLRK79b] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Discrete Optimization II*, volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier, 1979. doi:10.1016/S0167-5060(08)70356-X.

[GLS81] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981. doi:10.1007/BF02579273.

[GMUX17] V. Gupta, B. Moseley, M. Uetz, and Q. Xie. Stochastic online scheduling on unrelated machines. In *IPCO*, volume 10328 of *Lecture Notes in Computer Science*, pages 228–240. Springer, 2017. doi:10.1007/978-3-319-59250-3_19.

[GMUX20] V. Gupta, B. Moseley, M. Uetz, and Q. Xie. Greed works - online algorithms for unrelated machine stochastic scheduling. *Math. Oper. Res.*, 45(2):497–516, 2020. doi:10.1287/moor.2019.0999.

[GNYZ02] J. A. Garay, J. Naor, B. Yener, and P. Zhao. On-line admission control and packet scheduling with interleaving. In *INFOCOM*, pages 94–103. IEEE Computer Society, 2002. doi:10.1109/INFCOM.2002.1019250.

[Gol03] M. H. Goldwasser. Patience is a virtue: The effect of slack on competitiveness for admission control. *J. Sched.*, 6(2):183–211, 2003. doi:10.1023/A:1022994010777.

[Gra69] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–429, 1969. doi:10.1137/0117039.

[Gut13] A. Gut. *Probability: A Graduate Course*. Springer, 2 edition, 2013. doi:10.1007/b138932.

[HdLT01] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

[Hen18] M. Henzinger. The state of the art in dynamic graph algorithms. In *SOFSEM*, volume 10706 of *Lecture Notes in Computer Science*, pages 40–44. Springer, 2018. doi:10.1007/978-3-319-73117-9_3.

[HK99] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999. doi:10.1145/320211.320215.

[HS87] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, 1987. doi:10.1145/7531.7535.

[IC03] J. F. R. III and R. Chandrasekaran. Improved bounds for the online scheduling problem. *SIAM J. Comput.*, 32(3):717–735, 2003. doi:10.1137/S0097539702403438.

[IK75] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975. doi:10.1145/321906.321909.

[IL98] Z. Ivkovic and E. L. Lloyd. Fully dynamic algorithms for bin packing: Being (mostly) myopic helps. *SIAM J. Comput.*, 28(2):574–611, 1998. doi:10.1137/S0097539794276749.

[IM18] S. Im and B. Moseley. General profit scheduling and the power of migration on heterogeneous machines. In *SPAA*, volume 10807 of *Lecture Notes in Computer Science*, pages 755–776. Springer, 2018. doi:10.1007/978-3-319-77404-6_55.

[IMP15] S. Im, B. Moseley, and K. Pruhs. Stochastic scheduling of heavy-tailed jobs. In *STACS*, volume 30 of *LIPIcs*, pages 474–486. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.STACS.2015.474.

[IW91] M. Imase and B. M. Waxman. Dynamic steiner tree problem. *SIAM J. Discret. Math.*, 4(3):369–384, 1991. doi:10.1137/0404033.

[Jan09] K. Jansen. Parameterized approximation scheme for the multiple knapsack problem. *SIAM J. Comput.*, 39(4):1392–1412, 2009. doi:10.1137/080731207.

[Jan10] K. Jansen. An EPTAS for scheduling jobs on uniform processors: Using an MILP relaxation with a constant number of integral variables. *SIAM J. Discrete Math.*, 24(2):457–485, 2010. doi:10.1137/090749451.

[Jan12] K. Jansen. A fast approximation scheme for the multiple knapsack problem. In *SOFSEM*, volume 7147 of *Lecture Notes in Computer Science*, pages 313–324. Springer, 2012. doi:10.1007/978-3-642-27660-6_26.

[Jin19] C. Jin. An improved FPTAS for 0-1 knapsack. In *ICALP*, volume 132 of *LIPIcs*, pages 76:1–76:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.76.

[JK19] K. Jansen and K. Klein. A robust AFPTAS for online bin packing with polynomial migration. *SIAM J. Discret. Math.*, 33(4):2062–2091, 2019. doi:10.1137/17M1122529.

[JKK05] N. Johnson, A. Kemp, and S. Kotz. *Binomial Distribution*. John Wiley & Sons, Inc., 3 edition, 2005. doi:10.1002/0471715816.

[Joh54] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68, 1954. doi:10.1002/nav.3800010110.

[JS18] S. Jäger and M. Skutella. Generalizing the Kawaguchi-Kyan Bound to Stochastic Parallel Machine Scheduling. In *STACS*, volume 96 of *LIPIcs*, pages 43:1–43:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.STACS.2018.43.

[JSS20] S. Jamalabadi, C. Schwiegelshohn, and U. Schwiegelshohn. Commitment and slack for online load maximization. In *SPAA*, pages 339–348. ACM, 2020. doi:10.1145/3350755.3400271.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.

[Kel99] H. Kellerer. A polynomial time approximation scheme for the multiple knapsack problem. In *RANDOM-APPROX*, volume 1671 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 1999. doi:10.1007/978-3-540-48413-4_6.

[KK82] N. Karmarkar and R. M. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *FOCS*, pages 312–320. IEEE Computer Society, 1982. doi:10.1109/SFCS.1982.61.

[KP01] B. Kalyanasundaram and K. Pruhs. Eliminating migration in multi-processor scheduling. *J. Algorithms*, 38(1):2–24, 2001. doi:10.1006/jagm.2000.1128.

[KP03] B. Kalyanasundaram and K. Pruhs. Maximizing job completions online. *J. Algorithms*, 49(1):63–85, 2003. doi:10.1016/S0196-6774(03)00074-9.

[KP04] H. Kellerer and U. Pferschy. Improved dynamic programming in connection with an FPTAS for the knapsack problem. *J. Comb. Optim.*, 8(1):5–11, 2004. doi:10.1023/B:JOCO.0000021934.29833.6b.

[KPP04] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer, 2004. doi:10.1007/978-3-540-24777-7.

[KRT00] J. M. Kleinberg, Y. Rabani, and É. Tardos. Allocating bandwidth for bursty connections. *SIAM J. Comput.*, 30(1):191–217, 2000. doi:10.1137/S0097539797329142.

[KS94] G. Koren and D. E. Shasha. MOCA: A multiprocessor on-line competitive algorithm for real-time system scheduling. *Theor. Comput. Sci.*, 128(1&2):75–97, 1994. doi:10.1016/0304-3975(94)90165-1.

[KS95] G. Koren and D. E. Shasha. D^{over}: An optimal on-line scheduling algorithm for overloaded uniprocessor real-time systems. *SIAM J. Comput.*, 24(2):318–339, 1995. doi:10.1137/S0097539792236882.

[KV02] B. Korte and J. Vygen. *Combinatorial Optimization*. Springer, 2002. doi:10.1007/978-3-662-56039-6.

[Lab13] B. Labonté. Ein Simulationssystem für stochastische Scheduling-Probleme und empirische Untersuchung zur Approximationsgüte von Politiken. Master's thesis, Technische Universität Berlin, 2013.

[Law79] E. L. Lawler. Fast approximation algorithms for knapsack problems. *Math. Oper. Res.*, 4(4):339–356, 1979. doi:10.1287/moor.4.4.339.

[Lee03] J. Lee. Online deadline scheduling: multiple machines and randomization. In *SPAA*, pages 19–23. ACM, 2003. https://doi.org/10.1145/777412.777416.

[Leu04] J. Y. Leung, editor. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004. http://www.crcnetbase.com/isbn/978-1-58488-397-5.

[LMNY13] B. Lucier, I. Menache, J. Naor, and J. Yaniv. Efficient online scheduling for deadline-sensitive jobs: Extended abstract. In *SPAA*, pages 305–314. ACM, 2013. doi:10.1145/2486159.2486187.

[LWF96] J. Liebeherr, D. E. Wrege, and D. Ferrari. Exact admission control for networks with a bounded delay service. *IEEE/ACM Trans. Netw.*, 4(6):885–901, 1996. doi:10.1109/90.556345.

[Mol19] M. Molinaro. Stochastic ℓp load balancing and moment problems via the l-function method. In *SODA*, pages 343–354. SIAM, 2019. doi:10.1137/1.9781611975482.22.

[MP04] R. Mansini and U. Pferschy. Securitization of financial assets: Approximation in theory and practice. *Comput. Optim. Appl.*, 29(2):147–171, 2004. doi:10.1023/B:COAP.0000042028.93872.b9.

[MR85] R. H. Möhring and F. J. Radermacher. Introduction to stochastic scheduling problems. In K. Neumann and D. Pallaschke, editors, *Contributions to Operations Research*, pages 72–130. Springer, 1985. doi:10.1007/978-3-642-46534-5_6.

[MRW84] R. H. Möhring, F. J. Radermacher, and G. Weiss. Stochastic scheduling problems I - general strategies. *Z. Oper. Research*, 28(7):193–260, 1984. doi:10.1007/BF01919323.

[MRW85] R. H. Möhring, F. J. Radermacher, and G. Weiss. Stochastic scheduling problems II - set strategies-. *Z. Oper. Research*, 29(3):65–104, 1985. doi:10.1007/BF01918198.

[MSU99] R. Möhring, A. Schulz, and M. Uetz. Approximation in stochastic scheduling: The power of LP-based priority policies. *J. ACM*, 46(6):924–942, 1999. doi:10.1145/331524.331530.

[MSVW16] N. Megow, M. Skutella, J. Verschae, and A. Wiese. The power of recourse for online MST and TSP. *SIAM J. Comput.*, 45(3):859–880, 2016. doi:10.1137/130917703.

[MT90] S. Martello and P. Toth. Lower bounds and reduction procedures for the bin packing problem. *Discret. Appl. Math.*, 28(1):59–70, 1990. doi:10.1016/0166-218X(90)90094-S.

[MUV06] N. Megow, M. Uetz, and T. Vredeveld. Models and algorithms for stochastic online scheduling. *Math. Oper. Res.*, 31(3):513–525, 2006. doi:10.1287/moor.1060.0201.

[MV14] N. Megow and T. Vredefeld. A tight 2-approximation or preemptive stochastic scheduling. *Math. Oper. Res.*, 39(4):1297–1310, 2014. doi:10.1287/moor.2014.0653.

[MWW19] M. Mucha, K. Wegrzycki, and M. Wlodarczyk. A subquadratic approximation scheme for partition. In *SODA*, pages 70–88. SIAM, 2019. doi:10.1137/1.9781611975482.5.

[Oli82] H. J. Olivié. A new class of balanced search trees: Half balanced binary search trees. *RAIRO Theor. Informatics Appl.*, 16(1):51–71, 1982. doi:10.1051/ita/1982160100511.

[Pin16] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer International Publishing, 5 edition, 2016. doi:10.1007/978-3-319-26580-3.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[PS10] K. Pruhs and C. Stein. How to schedule when you have to buy your energy. In *APPROX*, volume 6302 of *Lecture Notes in Computer Science*, pages 352–365. Springer, 2010. doi:10.1007/978-3-642-15369-3_27.

[PST95] S. A. Plotkin, D. B. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.*, 20(2):257–301, 1995. doi:10.1287/moor.20.2.257.

[PST04] K. Pruhs, J. Sgall, and E. Torng. Online scheduling. In *Handbook of Scheduling*. Chapman and Hall/CRC, 2004. http://www.crcnetbase.com/doi/abs/10.1201/9780203489802.ch15.

[PW87] M. Pinedo and G. Weiss. The "largest variance first" policy in some stochastic scheduling problems. *Oper. Res.*, 35(6):884–891, 1987. doi:10.1287/opre.35.6.884.

[Ram92] B. Ram. The pallet loading problem: A survey. *International Journal of Production Economics*, 28(2):217–225, 1992. doi:10.1016/0925-5273(92)90034-5.

[Rhe15] D. Rhee. Faster fully polynomial approximation schemes for knapsack problems. Master's thesis, Massachusetts Institute of Technology, 2015. http://hdl.handle.net/1721.1/98564.

[Rot66] M. H. Rothkopf. Scheduling with random service times. *Manag. Sci.*, 12(9):707–713, 1966. doi:10.1287/mnsc.12.9.707.

[Rot12] T. Rothvoß. The entropy rounding method in approximation algorithms. In *SODA*, pages 356–372. SIAM, 2012. doi:10.1137/1.9781611973099.32.

[Sch03] A. Schrijver. *Combinatorial Optimization*. Springer, 2003. https://www.springer.com/de/book/9783540443896.

[Sch08] A. S. Schulz. Stochastic online scheduling revisited. In *COCOA*, volume 5165 of *Lecture Notes in Computer Science*, pages 448–457, Berlin, 2008. Springer. doi:10.1007/978-3-540-85097-7_42.

[Sga96] J. Sgall. On-line scheduling. In *Online Algorithms*, volume 1442 of *Lecture Notes in Computer Science*, pages 196–231. Springer, 1996. doi:10.1007/BFb0029570.

[Smi56] W. E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3(1-2):59–66, 1956. doi:10.1002/nav.3800030106.

[SS16] C. Schwiegelshohn and U. Schwiegelshohn. The power of migration for online slack scheduling. In *ESA*, volume 57 of *LIPIcs*, pages 75:1–75:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ESA.2016.75.

[SSS09] P. Sanders, N. Sivadasan, and M. Skutella. Online scheduling with bounded migration. *Math. Oper. Res.*, 34(2):481–498, 2009. doi:10.1287/moor.1090.0381.

[SSU16] M. Skutella, M. Sviridenko, and M. Uetz. Unrelated machine scheduling with stochastic processing times. *Math. Oper. Res.*, 41(3):851–864, 2016. doi:10.1287/moor.2015.0757.

[ST85] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985. doi:10.1145/2786.2793.

[Sto13] A. L. Stolyar. An infinite server system with general packing constraints. *Oper. Res.*, 61(5):1200–1217, 2013. doi:10.1287/opre.2013.1184.

[SU05] M. Skutella and M. Uetz. Stochastic machine scheduling with precedence constraints. *SIAM J. Comput.*, 34(4):788–802, 2005. doi:10.1137/S0097539702415007.

[SV16] M. Skutella and J. Verschae. Robust polynomial-time approximation schemes for parallel machine scheduling with job arrivals and departures. *Math. Oper. Res.*, 41(3):991–1021, 2016. doi:10.1287/moor.2015.0765.

[Tar83] R. E. Tarjan. Updating a balanced search tree in O(1) rotations. *Inf. Process. Lett.*, 16(5):253–257, 1983. doi:10.1016/0020-0190(83)90099-6.

[Vaz01] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001. http://www.springer.com/computer/theoretical+computer+science/book/978-3-540-65367-7.

[Wag59] H. M. Wagner. An integer linear-programming model for machine scheduling. *Naval Research Logistics Quarterly*, 6(2):131–140, 1959. doi:10.1002/nav.3800060205.

[Wal88] J. Walrand. *An Introduction to Queueing Networks*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Web82] R. R. Weber. Scheduling jobs by stochastic processing requirements on parallel machines to minimize makespan or flowtime. *J. Appl. Probab.*, 19(1):167–182, 1982. doi:10.2307/3213926.

[Wei66] H. M. Weingartner. Capital budgeting of interrelated projects: Survey and synthesis. *Manag. Sci.*, 12(7):485–516, 1966. doi:10.1287/mnsc.12.7.485.

[Wes00] J. R. Westbrook. Load balancing for response time. *J. Algorithms*, 35(1):1–16, 2000. doi:10.1006/jagm.2000.1074.

[WS11] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011. http://www.cambridge.org/de/knowledge/isbn/item5759340/?site_locale=de_DE.

[WVW86] R. Weber, P. Varaiya, and J. Walrand. Scheduling jobs with stochastically ordered processing times on parallel machines to minimize expected flowtime. *Journal of Applied Probability*, 23:841–847, 1986. doi:10.2307/3214023.

[Yan17] J. Yaniv. *Job Scheduling Mechanisms for Cloud Computing*. PhD thesis, Technion, Israel, 2017. http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2017/PHD/PHD-2017-03.pdf.

Zusammenfassung

Unsicherheit ist allgegenwärtig beim Lösen von praxisbezogenen Optimierungsproblemen und stellt eine große Herausforderung dar, wenn Resultate der kombinatorischen Optimierung in Handlungsempfehlungen für reale Anwendungen übersetzt werden sollen, da sich Problemparameter häufig ändern. Dadurch ist es sehr wahrscheinlich, dass die aktuelle Lösung unter den gegebenen Umständen gut funktioniert. Sobald jedoch die Inputdaten geringfügig geändert oder neue Informationen bekannt werden, ist dies meist nicht mehr der Fall und neue Ansätze müssen entwickelt werden.

Diese Arbeit beschäftigt sich mit Algorithmen, die selbst unter Unsicherheit beweisbar gute Lösungen finden, und konzentriert sich dabei auf zwei fundamentale Gebiete der kombinatorischen Optimierung: Packungs- und Schedulingprobleme. Von Packungsproblemen spricht man im Allgemeinen dann, wenn gewisse Objekte Behältern mit beschränkten Kapazitäten so zugewiesen werden sollen ohne diese zu überschreiten. Schedulingprobleme beschreiben die zeitliche Zuordnung von Aufgaben zu knappen Ressourcen oder Maschinen. Wir betrachten drei verschiedene Arten von Unsicherheit und die zugehörigen mathematischen Modelle: stochastische Informationen, online Modelle und dynamische Probleme.

Liegen Informationen über einen Problemparameter lediglich als stochastische Zufallsvariablen vor, so spricht man von stochastischen Informationen. Diese modellieren die Möglichkeit, Wissen aus vorherigen, ähnlichen Problemen oder Probleminstanzen zu nutzen, um neue, unbekannte Projekte zu realisieren. Dabei werden problemrelevante Parameter, wie die Dauer einer Aufgabe oder die Größe eines Objekts, als Zufallsvariablen modelliert und zu Beginn sind lediglich die zugrunde liegenden Wahrscheinlichkeitsverteilungen bekannt. Erst im Laufe der Zeit erfährt der Planer die Realisierung der Zufallsvariable und kann darauf basierend die Planung anpassen.

Online Modelle werden verwendet, wenn kaum Wissen über die Probleminstanz bekannt ist und dieses erst im Laufe der Planung verfügbar wird. Ein Planer muss auf der ihm zur Verfügung stehenden Datengrundlage gute (und manchmal unwiderrufbare) Entscheidungen treffen. Das Bekanntwerden von problemrelevanten Informationen kann dabei entweder schrittweise oder zu dem Planer unbekannten Zeitpunkten erfolgen.

Dynamische Probleme bilden die sich ständig verändernde Wirklichkeit ab, indem in jeder Runde die Probleminstanz lokalen Änderungen unterliegt. Solche lokale Änderungen können das Hinzufügen oder Entfernen von sowohl Objekten oder Aufgaben als auch von Behältern oder Maschinen sein. Die Schwierigkeit für dynamische Algorithmen liegt darin begründet, dass sie ihre Lösung zwar den Umständen anpassen können, diese jedoch schnellstmöglich berechnet werden muss.

In dieser Arbeit betrachten wir ein Schedulingmodell mit stochastischen Informationen, bei dem der erwartete durchschnittliche Fertigstellungszeitpunkt der Aufgaben minimiert werden soll. Wir schließen Gütegarantien, die unabhängig von der den stochastischen Informationen zugrunde liegenden Verteilungen sind, für sogenannte index-basierte Politiken mit Hilfe einer einfachen Klasse von Instanzen aus. Für etwas allgemeinere Politiken gelingt es uns für diese

Klasse von Instanzen eben solche Gütegarantien zu zeigen.

Wir behandeln des Weiteren ein Online-Schedulingmodell, bei dem die Aufgaben schrittweise nacheinander bekannt werden und lediglich von einigen der gegebenen Maschinen bearbeitet werden können. Unmittelbar bei Bekanntwerden einer neuen Aufgabe muss diese einer Maschine zugewiesen werden. Das Ziel ist, die Maximallast der Maschinen zu minimieren. Wir analysieren einen Online-Algorithmus, der Entscheidungen in begrenztem Maße widerrufen kann und dadurch eine gute Planung ermöglicht.

Zusätzlich untersuchen wir ein online-Schedulingmodell, bei dem die Aufgaben jeweils erst zu ihren Ankunftszeiten bekannt werden und dann von allen Maschinen bearbeitet werden können. Das Ziel ist es, die maximale Anzahl an Aufgaben vor ihren jeweiligen Deadlines fertigzustellen. Wir entwickeln hierfür einen Online-Algorithmus und analysieren seine Gütegarantie. Außerdem zeigen wir, dass unser Algorithmus bestmöglich ist. In anderen Worten, kein Algorithmus, der die Informationen im Laufe der Planung erhält, kann bessere Entscheidungen treffen.

In diesem Modell betrachten wir auch die Auswirkungen von verbindlichen Fertigstellungszusagen des Planers. Genauer gesagt untersuchen wir, wie sich die Gütegarantien von Online-Algorithmen verhalten, wenn der Planer zu einem bestimmten Zeitpunkt garantieren muss, dass eine bestimmte Aufgabe pünktlich fertiggestellt wird. Überraschenderweise gelingt es uns zu zeigen, dass einige moderate Fertigstellungszusagen keine allzu drastischen Auswirkungen auf die Performance von Online-Algorithmen haben. Wenn die Zusage bei Ankunft der Aufgabe gegeben werden muss, schließen wir die Existenz von guten Online-Algorithmen aus.

Zuletzt betrachten wir ein dynamisches Packungsproblem, bei dem sowohl die Menge der Objekte als auch die Behälter lokalen Änderungen unterliegen: In jeder Runde wird entweder ein neues Objekt oder ein neuer Behälter hinzugefügt oder ein Objekt oder ein Behälter entfernt. Jedes Objekt ist durch seine Größe sowie seinen Wert charakterisiert. Das Ziel ist es hierbei, Objekte von maximalem Gesamtwert zu packen ohne die Kapazitäten der Behälter zu überschreiten. In diesem Modell darf sich die Lösung eines Algorithmus‘ den Umständen anpassen und beliebig verändern, solange die neue Lösung in sublinearer Zeit berechnet werden kann. Wir beschreiben hierfür einen Algorithmus, der nahezu optimale Lösungen liefert.