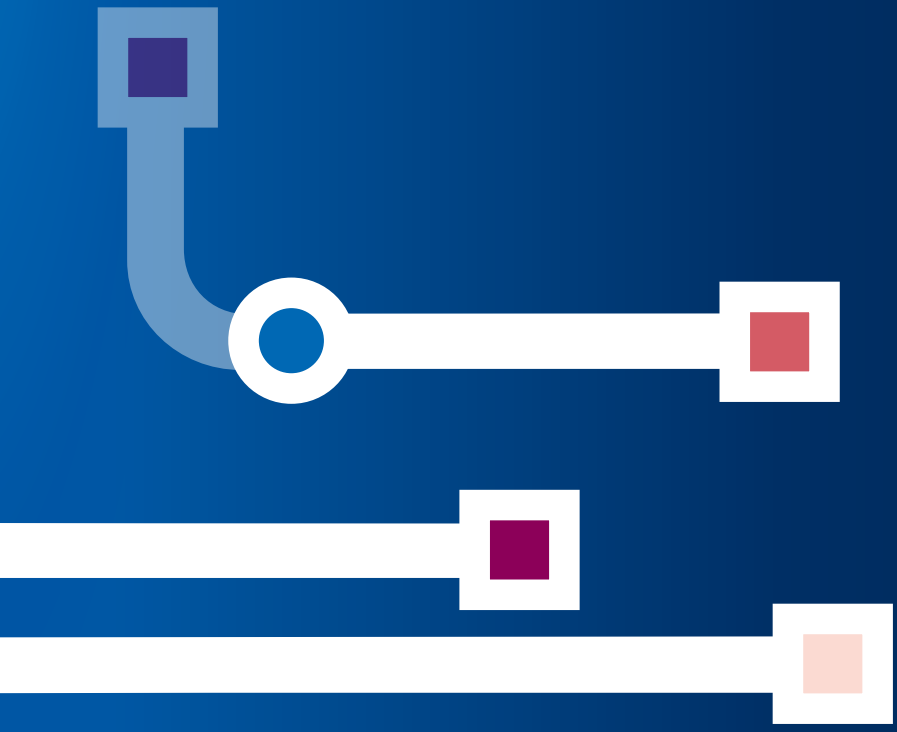
Matching and Packing Problems

Optimization under Uncertainty in Theory and Practice

Lukas Nölke



Matching and Packing Problems

Optimization Under Uncertainty in Theory and Practice

vorgelegt von

Lukas Nölke M.Sc.

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

Doktor der Naturwissenschaften
– Dr. rer. nat. –

genehmigte Dissertation

*Gutachter*innen:*

Prof. Dr. Nicole Megow
Fachbereich 3 - Mathematik und Informatik
Universität Bremen

Prof. Dr. Sven O. Krumke
Fachbereich Mathematik
RPTU Kaiserslautern-Landau

Tag der wissenschaftlichen Aussprache: 21. Februar 2023

Bremen, 2023

Lukas Nölke M.Sc.

Matching and Packing Problems – Optimization Under Uncertainty in Theory and Practice

Genehmigte Dissertation, Bremen, 2023

Gutachter*innen: Prof. Dr. Nicole Megow und Prof. Dr. Sven O. Krumke

Universität Bremen

Fachbereich 3 - Mathematik und Informatik

Combinatorial Optimization and Logistics Group (CSlog)

Bibliothekstr. 5

28359 Bremen

Universität
Bremen

Acknowledgements

This thesis is the product of many years of research, riddled with ups and downs, frustration and laughter, insights and dead ends. In all of this, I was accompanied by a wonderful cast of people that encouraged me, led me by example, or sometimes were simply there for me. I wish to deeply thank all of them for their support and contribution; this thesis would not have been possible without them. The following people had a particularly formative influence and merit specific mention.

First and foremost, I thank my advisor Nicole Megow for introducing me into the exciting world of combinatorial optimization and its terrific research community, and for welcoming me into the Combinatorial Optimization and Logistics (CSLog) group, which I came to experience as an open, inspiring, and nurturing environment, both scientifically and socially. I am particularly grateful for her supportive yet empowering style, the many research conversations that never failed to somehow nudge me in the right direction, and her encouragement to attend workshops, summer schools, and conferences, and to collaborate internationally. I also want to thank Sven Krumke for taking on the second assessment of this thesis.

By far the favorite part of my PhD time were the hours spent in front of a whiteboard, developing ideas and discussing problems together with my incredible colleagues, to whom I am immensely grateful. In particular, I thank the members of the CSLog group for the fantastic working atmosphere and Antonios Antoniadis, Parinya Chalermsook, Peter Kling, José Verschae, and Andreas Wiese for inviting me to wonderful workshops and research visits. I owe special gratitude to Franziska Eberle, with whom I was lucky enough to share an office during most of our time as PhD students. It was a joy to bounce around ideas, share in each other's successes and frustrations, and "prove" that $\mathcal{P} = \mathcal{NP}$; thank you for always having my back.

I also want to thank Martin Böhm, Franziska Eberle, Felix Hommelsheim, Alexander Lindermayr, Björn Ludwig, Jan Cedric Mertens, Kevin Schewior, and Jens Schlöter for proofreading different parts of this thesis and for their help in preparing for my defense.

Moreover, I wish to express my deep gratitude to all my friends, for motivating me and keeping my spirits up during the last years. To Jan, for cutting me off from playing games when I was in danger of procrastinating; to Giacomo, for cutting me off from work and going for a walk in Bürgerpark when I was in danger of not getting enough sunlight; and to Thai, for making sure I eat and sleep enough, and for supporting and enduring me even in difficult times.

Finally, I want to thank my family, especially my parents and my brother, for their unconditional support, encouragement, and unfaltering belief in me. It means the world.

Bremen, March 2023 *Lukas Nölke*

Table of Contents

Introduction

When considering optimization problems that arise from real-world decision-making processes, uncertainty is a ubiquitous phenomenon that poses a significant obstacle. Be it a baker, that needs to decide how much pastry to produce without knowing what his customers will order; or a data center, in which computational resources need to be allocated subject to various uncertainties, ranging from unknown future workload to unforeseen power fluctuations; when solving optimization problems in the real world it often is unavoidable to make decisions without knowing what their full effects will be. This thesis investigates how to algorithmically deal with such uncertainties when solving matching and packing problems.

Matching problems are well-studied and among the most fundamental problems in combinatorial optimization. The task here is to compute from a given set of items a collection of disjoint pairs – a *matching* – while optimizing some objective. Moreover, there may be additional constraints, e.g., rules restricting which of the items can be paired. For instance, in the *minimum-cost bipartite matching* problem, which will be of particular interest in this thesis, an item is said to be either request or server and a pair in the matching must contain one of each. Further, each pair is associated with some cost, and the optimization objective is to minimize the sum of costs of all pairs in the matching. Matching problems arise wherever a rational allocation of resources is paramount, for instance, in kidney exchange programs, online ad allocation, or when assigning virtual machines to physical hosts in cloud computing. Besides the abundance of practical applications, matchings also play a crucial role as a tool to solve other complex optimization problems, for instance, as a subroutine in the Christofides Algorithm for the traveling salesperson problem [Chr22].

Packing problems, the second area of our interest, form an equally fundamental class of optimization problems. Generally speaking, these are problems in which items need to be assigned to containers with limited capacities while optimizing some objective. The goal might be to pack a given set of items into as few containers as possible, or, as in the *multiple knapsack* problem, to

pack only a subset of items into a limited amount of containers with varying capacities to maximize the total value of packed items. Such tasks arise naturally in many logistics applications. For instance, when packing items into freight containers and those then onto cargo ships, or when sensibly packing your suitcase before embarking on a journey. Other applications of packing problems include cutting pieces of material into predefined shapes while wasting as little material as possible, financial modeling, energy management, software resource management, and yield management for airlines, hotels, and car rentals.

Most packing problems are $\mathcal{NP}$-hard. This means that with computational means available today, we cannot expect to obtain optimal solutions for these problems in reasonable time. Formally, reasonable time means that the running time of an algorithm is polynomial in the size of the input. Practically speaking, this means that exact algorithms for $\mathcal{NP}$-hard problems might not find a solution until long after we actually need it. Nevertheless, since these problems are so omnipresent and the need to solve them is inescapable, we may not give up the quest for efficient algorithms. Rather, we compromise and strive for efficient algorithms that produce solutions that come as close as possible to being optimal. An algorithm that consistently, for every instance, produces in polynomial time a solution whose objective value is within an α factor of that of an optimal solution is called an *α-approximation algorithm*. Interestingly, for many $\mathcal{NP}$-hard problems, e.g., for multiple knapsack, it is possible to design α-approximation algorithms with α arbitrarily close to 1. Although the running time grows rather fast when approaching this limit, the solution quality is extremely close to being optimal.

In contrast, minimum-cost bipartite matching and many other matching problems can be solved to optimality in polynomial time. However, as soon as uncertainty comes into play, a problem may become exceedingly more difficult to deal with and it may no longer be possible to obtain optimal solutions. When speaking of optimal solutions in the context of uncertainty, we always mean an optimal solution for the same problem while assuming that no uncertainty is present and all information about the problem is readily available. We also refer to this as an (optimal) *offline* solution. If the input of a problem is revealed only incrementally over time and an algorithm has to react to incoming information immediately, irrevocably, and without

knowledge of the future, we speak of an *online problem* and *online algorithm*, respectively. An online algorithm is said to be α-*competitive*, if it obtains for every instance a solution whose objective value is within a factor of α of that of an offline solution. Unlike in the case of approximation algorithms, we do not require online algorithms to run in polynomial time and focus instead on the information-theoretical question of how the lack of complete information impacts the solution quality. It should come as no surprise that missing information about crucial problem parameters generally prevents us from reaching the same quality as an optimal offline solution. Consider, for instance, the minimum-cost bipartite matching problem in the online setting in which the set of servers is known but the requests appear one by one and on arrival have to be immediately and irrevocably matched to a free server. It was shown that there cannot exist an α-competitive algorithm for *any* $\alpha \in \mathbb{N}$, and even when restricting to the case of metric costs, there is only a $(2n-1)$-competitive algorithm, which is best possible [KP93; KMV94].

Online problems are only one of several ways to model uncertainty. Recognizing that the irrevocability of an online algorithm's decision is a rather strong restriction that might not always reflect reality, the *recourse setting* allows for decisions to be revoked but imposes a budget on the number of changes that an algorithm may perform. Similarly, in the migration setting, there is a bound not on the number of changes but instead on their total volume, which quantifies the severity of the undergone changes. *The dynamic setting*, where real-time responsiveness is critical, completely removes the irrevocability but requires the current solution to be recomputed within a limited time frame whenever information about the problem is revealed or altered. Streaming problems instead limit the memory that is used to store past inputs, as necessary when, for instance, computing on enormous amounts of data, as in social networks. Other models of uncertainty include stochastic information, which describes information that is unknown but follows a known probability distribution; explorable uncertainty, where one pays to reveal uncertain parameters and the goal is to minimize the cost necessary to find a satisfactory solution with certainty; and robust optimization, where the constraints and the objective function come from a known set of possibilities and the task is to compute a solution that is feasible for all possible constraints and optimal for the worst-case objective.

For all these settings, theoretical research traditionally evaluates algorithms using rigorous worst-case measures, as described above for approximation and online algorithms, by requiring an algorithm to satisfy certain bounds for *all* instances of a problem. This is the gold standard in theoretical evaluation and allows for very strong statements. When considering real-world applications, such hard guarantees are particularly important for safety-critical tasks, for instance, in flight and space travel, medical applications, or autonomous vehicles. Moreover, the pursuit of worst-case instances that constitute lower bounds often reveals insights into the nature of a problem and inspires novel algorithmic approaches. On the other hand, such worst-case instances may not (often) occur in practice for some non-safety-critical problems. As a consequence, seemingly bad algorithms – at least according to the previously described evaluations – may perform remarkably well here. Indeed, when solving real-world problems, it is often more sensible to evaluate algorithms on an average-case basis for a collection of typical instances. Nevertheless, it is important to use a comparison to optimal solutions or suitable lower bounds thereon in order to meaningfully assess an algorithm's performance.

Outline and Bibliographic Remarks

Chapter 1 In the first chapter, we consider the minimum-cost bipartite matching problem on the line metric. That is, every request or server is identified with a point on the real line and the cost of matching a request to a server is equal to their distance on the line. The objective is to minimize the total cost of the matching. In the online setting, n requests appear one by one and have to be matched immediately and irrevocably to a free server. Here, the best known algorithm is due to Raghvendra [Rag18] with a competitive ratio of $\Theta(\log n)$. Very recently, Peserico and Scquizzato [PS21] showed that no online algorithm can achieve a competitive ratio better than $\Omega(\sqrt{\log n})$ and solved the long standing open question whether or not there exists a constant-competitive online algorithm for this problem.

Our contribution in this chapter is a first investigation of online minimum-cost matching on the line in the *recourse setting*, where previously matched edges may be reassigned a limited number of times. In contrast to the pure online setting, we show that with an amortized recourse budget of $O(\log n)$, we

can indeed obtain a constant-competitive algorithm. This is one of the first non-trivial results for minimum-cost matching with recourse. We further refine this result to allow for a trade off between the competitive ratio and the recourse budget. Specifically, we show that, for any function $f : \mathbb{N} \longrightarrow \mathbb{N}$ with $1 \leq f(n) \leq \log n$, there is a $O(f(n))$-competitive algorithm with an amortized recourse budget of $O(\frac{\log n}{f(n)})$.

For so-called alternating instances, with no more than one request between two servers, we obtain a near-optimal result. We give a very simple algorithm that is $(1+\varepsilon)$-competitive and reassigns any request at most $O(\frac{1}{\varepsilon^2})$ times. This special case is interesting as a lower bound of $\Omega(\log n)$ [AFT18], which holds for a quite large class of online algorithms, including all deterministic algorithms in the literature, is constructed using such instances. As such, our result raises a cautious hope that it may be possible that also for general instances one can design algorithms with constant competitive ratio and only a constant amortized recourse budget.

Bibliographic Remark: Chapter 1 is based on joint work with Nicole Megow. An extended abstract has been published at APPROX 2020 [MN20]. A full version has been submitted to Algorithmica.

Chapter 2 The second chapter of this thesis considers knapsack problems. In multiple knapsack, we are given multiple knapsacks with different capacities and items with values and sizes. The task is to find a packing of a subset of the items into the knapsacks without exceeding their capacities such that the total value of packed items is maximized. We investigate this problem and special cases thereof in the *dynamic setting*. That is, our goal is to handle the arrival and departure of individual items or knapsacks during the execution of the algorithm with worst-case update time polylogarithmic in the current number of items. 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).

Motivated by the theoretical interest in knapsack problems and their practical relevance, we aim to bridge this gap. Specifically, we give a fully dynamic algorithm for multiple knapsack that maintains $(1-\varepsilon)$-approximate solutions with an update time of approximately $(\frac{1}{\varepsilon} \cdot \log n)^{O(\frac{1}{\varepsilon})}$ and justify the

superpolynomial dependency on ε under the assumption that $\mathcal{P} \neq \mathcal{NP}$. To facilitate this running time, we maintain solutions implicitly as a data structure that supports query operations that return the computed solution value and the packing of any queried item in polylogarithmic time. In order to solve multiple knapsack, we show how to decompose the problem into two subproblems which are of independent interest and for which we provide even faster running times: multiple knapsack with the number of knapsacks polylogarithmic in n, and multiple knapsack with an additional set of $(\frac{\log n}{\varepsilon})^{\Theta(1)}$ knapsacks that the optimal solution we evaluate our algorithm against may not use; this is also called resource augmentation. Moreover, we present a particularly efficient algorithm for the knapsack problem with an update time that is polynomial in n and $\frac{1}{\varepsilon}$ and query times that are comparable to those of accessing an explicit solution from memory. Maybe surprisingly, for all cases considered, we recompute a solution from scratch for every update and only require the storage of our items and knapsacks in suitably designed data structures. Importantly, this shows that, to compute a solution for these knapsack problems, we do not need exact knowledge about the whole input, but only a small amount of information of polylogarithmic size.

Bibliographic Remark: Chapter 2 is based on joint work with Franziska Eberle, Nicole Megow, Bertrand Simon, and Andreas Wiese. An extended abstract has been published at FSTTCS 2021 [EMN+21]. Parts of it, in particular Sections 2.4 and 2.5, also appear in the PhD thesis by Franziska Eberle [Ebe20].

Chapter 3 This chapter investigates the *anchored rectangle packing (ARP)* problem. Here, we are given a set of points P in the unit square $[0,1]^2$ and seek a maximum-area set $\mathcal{S}$ of axis-aligned interior-disjoint rectangles, each of which is anchored at a point $p \in P$. That is, p lies at some specific position on the rectangle. In the most prominent variant, the *lower-left-anchored rectangle packing (LLARP)* problem, rectangles are anchored in their lower-left corner. Freedman [Tut69, Unsolved Problem 11, page 345] conjectured in 1969 that, if $(0,0) \in P$, then there is a LLARP that covers an area of at least 0.5. Somewhat surprisingly, this conjecture remains open to this day, with the best-known result covering an area of 0.39 [DKK+21].

For the LLARP problem, we investigate two different settings with resource augmentation: In the first, we allow an ε-perturbation of the input P, and develop an algorithm that covers at least as much area as an optimal solution of the original problem. In the second setting, we permit an ε-overlap between rectangles and give a $(1-\varepsilon)$-approximation. For the *center-anchored rectangle packing* problem, where rectangles are anchored in their center, we provide a polynomial time approximation scheme. In fact, our PTAS applies to any ARP problem where the anchor lies in the interior of the rectangles.

Bibliographic Remark: Chapter 3 is based on joint work with Antonios Antoniadis, Felix Biermeier, Andrés Cristi, Christoph Damerius, Ruben Hoeksma, Dominik Kaaser, and Peter Kling. An extended abstract has been published at ESA 2019 [ABC+19]. The PTAS for center-anchored rectangle packing will also appear in the PhD thesis by Christoph Damerius.

Chapter 4 In the final chapter, we consider a practical problem motivated by a project in the hot rolling mill of a multinational steel manufacturer. It concerns the allocation and maintenance of scarce resources in resource-constrained production environments. Specifically, we consider the problem of *simultaneous allocation and maintenance scheduling* (SAMS) of recyclable, heterogeneous resources. The task here is threefold: first, to allocate resources to production jobs in order to satisfy their resource demands; second, to schedule the maintenance of resources – according to the aforementioned allocation – on heterogeneous maintenance machines with resource-type dependent setup times; and third, to schedule the resource-constrained production jobs. The resource allocation determines a maintenance cost that depends on the profiles of the resource and the corresponding demand, and the maintenance scheduling incurs a waiting cost whenever the late maintenance of a resource delays the corresponding production job. A bi-criteria objective function is considered to simultaneously minimize the maintenance and waiting costs. An added difficulty lies in the online nature of the production sequence since at any time only the next k productions jobs and their properties are known, we call this *lookahead* k.

After introducing the SAMS problem, we develop an integer linear program (ILP) for SAMS in the offline setting and two simpler ILPs that lower bound the individual optimization objectives. We also discuss how the ILPs can be

used in the online setting with lookahead k. However, due to their computational complexity, the ILPs are mainly useful for production design and the evaluations of further algorithms for SAMS. To address this issue and provide a viable algorithmic approach for practitioners, we present an efficient heuristic that divides the SAMS problem into three stages and uses matching and scheduling techniques to obtain good-quality solutions for SAMS with lookahead k. The heuristic is evaluated against real-world datasets and the developed integer linear programs.

Bibliographic Remark: Chapter 4 is based on work with Alexander Lindermayr and Nicole Megow. At the time of writing, this work is unpublished.

Publications Underlying the Thesis

[ABC+19] A. Antoniadis, F. Biermeier, A. Cristi, C. Damerius, R. Hoeksma, D. Kaaser, P. Kling, and L. Nölke. "On the Complexity of Anchored Rectangle Packing". In: *ESA*. Vol. 144. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 8:1–8:14 (cit. on pp. 7, 97, 109).

[EMN+21] F. Eberle, N. Megow, L. Nölke, B. Simon, and A. Wiese. "Fully Dynamic Algorithms for Knapsack Problems with Polylogarithmic Update Time". In: *FSTTCS*. Vol. 213. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 18:1–18:17 (cit. on p. 6).

[MN20] N. Megow and L. Nölke. "Online Minimum Cost Matching with Recourse on the Line". In: *APPROX-RANDOM*. Vol. 176. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 37:1–37:16 (cit. on p. 5).

Publications by the Author Outside the Scope of this Thesis

[ACC+21] A. Antoniadis, M. Capretto, P. Chalermsook, C. Damerius, P. Kling, L. Nölke, N. O. Acosta, and J. Spoerhase. "On Minimum Generalized Manhattan Connections". In: *WADS*. Vol. 12808. Lecture Notes in Computer Science. Springer, 2021, pp. 85–100.

[BHM+22] M. Böhm, R. Hoeksma, N. Megow, L. Nölke, and B. Simon. "On Hop-Constrained Steiner Trees in Tree-Like Metrics". In: *SIAM J. Discret. Math.* 36.2 (2022), pp. 1249–1273.

[EHM+23] F. Eberle, R. Hoeksma, N. Megow, L. Nölke, K. Schewior, and B. Simon. "Speed-robust scheduling: sand, bricks, and rocks". In: *Math. Program.* 197.2 (2023), pp. 1009–1048.

[ELM+22] F. Eberle, A. Lindermayr, N. Megow, L. Nölke, and J. Schlöter. "Robustification of Online Graph Exploration Methods". In: *AAAI*. AAAI Press, 2022, pp. 9732–9740.

Online Minimum-Cost Matching with Recourse on the Line

1

Matching problems are among the most fundamental problems in combinatorial optimization, with great importance in both, theory and applications. In the *minimum-cost (bipartite) matching* problem, we are given a complete bipartite graph $G = (R \cup S, E)$ with positive edge costs $c(e) = c(s, r)$, for $e = (s, r) \in E$. Elements of R and S are called *requests* and *servers*, respectively, with $n := |R| \leq |S|$. A *matching* $M \subseteq E$ is a set of pairwise non-incident edges. M is called a *matching of* R if every request in R is matched to a server in S, i.e., if it is incident to exactly one edge of M. The minimum-cost matching problem asks to compute a matching of R with minimum cost, where the *cost* of a matching M is given by $c(M) := \sum_{e \in M} c(e)$. When all information is given in advance, the optimum can be computed efficiently, e.g., using the Hungarian Method by Kuhn and Yaw [KY55].

In the *online setting*, however, the set of requests is not known a priori. Requests arrive online, one by one, and have to be matched immediately and irrevocably to an unmatched server. As we cannot hope to find an optimal matching under these restrictions, we use standard competitive analysis to evaluate the performance of algorithms. An online matching algorithm is said to be α*-competitive* if it computes for any instance a matching M with $c(M) \leq \alpha \cdot \mathrm{OPT}$, where OPT denotes the cost of an optimal matching when knowing all requests from the start, i.e., for the *offline setting*. The *competitive ratio* of an algorithm is the smallest α for which it is α-competitive. Independently, Kalyanasundaram and Pruhs [KP93], and Khuller, Mitchell, and Vazirani [KMV94] showed that for arbitrary edge costs, the competitive ratio of any algorithm for the online minimum-cost matching problem is unbounded. They also developed the same deterministic algorithm, Permutation, which is $(2n - 1)$-competitive for metric costs and showed that this is optimal for deterministic online algorithms. A remarkable recent

result by Nayyar and Raghvendra [NR17] is a fine-grained analysis of an online algorithm based on *t-net cost* [Rag16] showing a competitive ratio of $O(\mu(G)\log^2 n)$, where $\mu(G)$ denotes the maximum ratio of the minimum TSP tour and the weighted diameter of a subset of G. This result is particularly interesting as it relates structural parameters of the metric space to the performance guarantee of the algorithm.

For a long time, the online minimum-cost matching problem has resisted all attempts for achieving an $O(1)$-competitive algorithm even for special metric spaces such as the line. In *online minimum-cost matching on the line*, the edge costs are induced by a line metric; that is, we identify each vertex of G with a point on the real line and the cost of an edge between a request and a server equals their distance on the line. The competitive ratio of the aforementioned algorithm in [NR17] is then $O(\log^2 n)$, as $\mu(G) = 2$. Subsequently, Raghvendra [Rag18] improved this to $\Theta(\log n)$. Only very recently, Peserico and Scquizzato [PS21] showed that no online algorithm can achieve a competitive ratio better than $\Omega(\sqrt{\log n})$. Thus, the quest for finding an $O(1)$-competitive algorithm, even a randomized one, is indeed hopeless, at least when restricted to the strict online setting.

In this chapter, we consider online minimum-cost matching on the line with *recourse*. In the recourse model, we allow to change a small number of past decisions, thereby relaxing the online model. Specifically, at any point, we may delete a set of edges $\{(r_i, s_i)\}_i$ of the current matching and rematch the requests r_i to different (free) servers. Online optimization with recourse is an alternative model to standard online optimization which has received increasing popularity recently. Obviously, if the recourse is not limited then one can just simulate an optimal offline algorithm, and the online nature of the problem disappears. We say an algorithm requires *amortized recourse budget* β if it rematches requests at most βn times in total. The challenging question for online minimum-cost matching on the line is whether it is possible to maintain an $O(1)$-competitive solution with bounded recourse, i.e., with sublinear recourse budget.

Our Results We answer this question to the affirmative and give non-trivial results for online minimum-cost matching with recourse. We show that with

limited recourse, one can indeed maintain a constant competitive solution on the line.

Theorem 1.1. *The online minimum-cost matching problem on the line admits an $O(1)$-competitive algorithm with amortized recourse budget $O(\log n)$.*

Our algorithm builds on the t-net-cost algorithm by Raghvendra [Rag16; Rag18]; details follow later. It has the nice property that it interpolates between an $O(\log n)$-competitive online solution (without any recourse) and an $O(1)$-approximate offline solution (with possibly large recourse). Our algorithm can further be adapted to allow for a range of different cost-recourse trade-offs. We observe that our analysis is asymptotically optimal.

Theorem 1.2. *For any function $f : \mathbb{N} \longrightarrow \mathbb{N}$ with $1 \leq f(n) \leq \log n$, there is an algorithm for online minimum-cost matching on the line that is $O(f(n))$-competitive and has an amortized recourse budget of $O(\frac{\log n}{f(n)})$.*

Additionally, we investigate a special class of instances, called *alternating* instances, where between any two requests on the line there is at least one server. This class is interesting as the quite strong lower bound of $\Omega(\log n)$, given by Antoniadis, Fischer, and Tönnis [AFT18], holds already on such simple instances. While it does not hold for arbitrary instances, the bound is true for all known deterministic online algorithms *without recourse*. For alternating instances, we present a more direct and near-optimal algorithm with a scalable performance-recourse trade-off.

Theorem 1.3. *For alternating instances of online minimum-cost matching on the line, there is a $(1 + \varepsilon)$-competitive algorithm that reassigns each request $O(\frac{1}{\varepsilon^2})$ times.*

While the algorithm is quite simple, the proof requires a clever charging scheme that exploits the special structure of optimal solutions on alternating instances. We observe that a large number of recourse actions for a specific request implies large edges in the optimal solution elsewhere on the line.

As a byproduct we give a simple analysis of (a variant of) the algorithm in the traditional online setting without recourse. We show that it has a

competitive ratio of $O(\min\{\log n, \log \Delta\})$ for instances with alternating requests on the line, where Δ is the ratio between the largest and shortest request-server distance, also called the aspect ratio of the metric. This result refines the competitive ratio $\Theta(\log n)$ of the currently best known online algorithm [Rag18].

Remark Simultaneously and independently of our work, Gupta, Krishnaswamy and Sandeep [GKS20] obtained a similar result for online minimum-cost matching with recourse on the line. Their algorithm builds on the $O(n)$-competitive Permutation algorithm [KP93; KMV94] and adapts it for the recourse setting. On the line, this is done by first matching edges according to Permutation and then asymmetrically applying recourse to arcs (r, s) of the current matching that overlap in a certain way. Both, their algorithm and analysis are completely different from ours. They further obtain a more general $O(\log n)$-competitive algorithm with amortized recourse $O(\log n)$ for arbitrary metrics. Note that Permutation in fact coincides with the t-net-cost algorithm for $t = 1$.

Further Related Work Extensive literature is devoted to online bipartite matching problems. The maximum matching variant is quite well understood. For the unweighted setting, Karp, R. Vazirani, and U. Vazirani [KVV90] gave optimal deterministic and randomized algorithms with competitive ratio 2 and $e/(e-1)$, respectively. The weighted maximization setting does not admit bounded guarantees in general, but intensive research investigates models with additional assumptions; see, for instance, the survey by Mehta [Meh13].

The online minimum-cost matching problem is much less understood. Despite recent advances, there remains a gap between the best known upper and lower bounds of $O(\log n)$ and $\Omega(\sqrt{\log n})$, respectively. Randomization allows an improvement upon the best-possible deterministic competitive ratio of $(2n-1)$ for metric online bipartite matching [KP93; KMV94]; there is an $O(\log^2 n)$-competitive randomized algorithm due to Bansal et al. [BBG+14]. On the line, no such improvement on the deterministic result by randomization is known; $O(\log n)$ is the best known competitive ratio for both, deterministic and randomized algorithms [Rag18; GL12].

Interestingly, Gairing and Klimm [GK19] showed that when assuming randomization in the order of request arrivals (instead of an adversarial arrival order), the natural Greedy algorithm that matches an arriving request to the closest *free*, i.e., unmatched, server is n-competitive for general metric spaces. Furthermore, the online *t-net-cost* algorithm is $O(\log n)$-competitive [Rag16] in this case. Very recently, Gupta et al. [GGP+19] gave an $O((\log \log \log n)^2)$-competitive algorithm in the model with online known i.i.d. arrivals.

Recourse models received quite some attention in the past decade. In particular, maintaining an online cardinality-maximal bipartite matching with recourse was studied extensively; see, e.g., [BHR19; BLS+14; CDK+09; GKK+95; ADJ18; SKL+20; BD20] and references therein. Bernstein et al. [BHR19] showed that the 2-competitive greedy algorithm uses amortized $O(n \log^2 n)$ reassignments, leaving a small gap to the lower bound of $\Omega(n \log n)$. In contrast, for the min-cost variant, it remained a challenging question whether recourse can improve upon the competitive ratio. Even on the line, it remained open whether and how recourse can improve the bound of $O(\log n)$ [Rag18].

The following two models address other types of matching with recourse. In a setting motivated by scheduling, several requests can be matched to the same server and the goal is to minimize the maximum number of requests assigned to a server. Gupta et al. [GKS14] achieve an $O(1)$-competitive ratio with amortized $O(n)$ edge reassignments. A quite different two-stage robust model has been proposed recently by Matuschke et al. [MSV19]. In a first stage, one must compute a perfect matching on a given graph, and, in a second stage, a batch of $2k$ new nodes appears, which must be incorporated into the first-stage solution to maintain a low-cost matching by reassigning only few edges. For matching on the line, they give an algorithm that maintains a 10-approximate matching and reassigns at most $2k$ edges.

Recourse in online optimization has been investigated also for other min-cost problems even though less than for maximization problems. Most notably seems the online minimum Steiner tree problem [GGK16; IW91; MSV+16; LOP+15]. Here, one edge reassignment per iteration suffices to maintain an $O(1)$-competitive algorithm [GGK16], whereas the online setting without recourse admits a competitive ratio of $\Omega(\log n)$.

The recourse model has some relation to *dynamic algorithms*. Instead of minimizing the number of past decisions that are changed (recourse), the dynamic model focuses on the running time to implement this change (*update time*). A full body of research exists on maximum (weighted) bipartite matching; we refer to the nice survey by Chaudhuri et al. [DP14]. We are not aware of any results for maintaining a minimum-cost matching.

1.1 Preliminaries

A path P is called *alternating* with respect to a matching M, if every other edge of P is contained in M. An alternating path is called *augmenting* with respect to M if it starts and ends at vertices not covered by M. A common method for increasing the cardinality of an existing matching M is to *augment* along an augmenting path P, resulting in a larger matching $\tilde{M}$ given by the symmetric difference[1] $M \oplus P$. There may be a choice between different augmenting paths; typically, a path of minimum cost (with respect to some metric) is selected. Recently, Raghvendra [Rag16] introduced the following metric. For $t > 1$, the *t-net cost* of a path P with respect to a matching M is defined as

$$\phi_t^M(P) := t \cdot c(P \setminus M) - c(P \cap M) = t \cdot c(P \cap \tilde{M}) - c(P \cap M). \quad (1.1)$$

Our algorithm maintains three different matchings: the *recourse matching* M, which is the actual output of the algorithm, and two auxiliary matchings based on (online and offline versions of) the t-net-cost algorithm [Rag16], namely, the *offline matching* M^* and the *online matching* M'. While M^* is a near-optimal offline matching that possibly requires a large amount of recourse, M' is an online matching that is $O(\log n)$-competitive [Rag18] but uses no recourse. We describe how M^* and M' are obtained based on the above cost function; see also [Rag16; NR17; Rag18]. When speaking of the matching M_i, M_i^* or M_i', we refer to the state of the respective matching after serving the i-th request.

[1] For two sets X, Y, their symmetric difference is given by $X \oplus Y := (X \cup Y) \setminus (X \cap Y)$. For matchings M_1, M_2, their symmetric difference $M_1 \oplus M_2$ consists of disjoint paths and cycles, whose edges are alternating between M_1 and M_2.

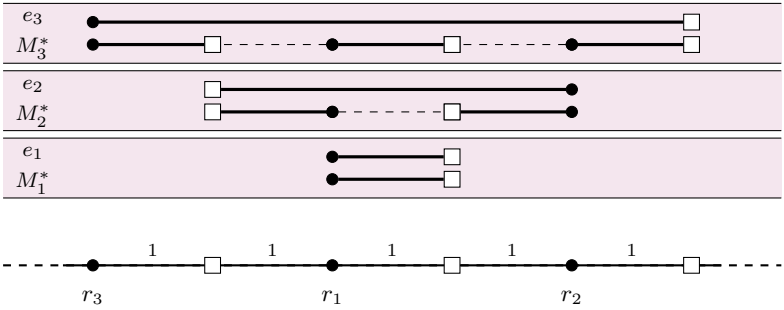
Upon arrival of the i-th request r_i, the *offline t-net-cost algorithm* constructs the offline matching M_i^* by augmenting M_{i-1}^* along an alternating path P_i of minimum t-net cost with respect to M_{i-1}^*. That is, $M_i^* := M_{i-1}^* \oplus P_i$. By definition, this path starts at r_i and ends at a free server, which we denote by s_i. While this procedure may require a large amount of recourse, the resulting matching has been shown to have bounded cost.

Lemma 1.4 (Raghvendra [Rag16]). *For any $t > 1$, we have $c(M_i^*) \leq t \cdot \mathrm{OPT}_i$ for all $i \in \{1, 2, \ldots, n\}$, where $\mathrm{OPT}_i := c(M_i^{\mathrm{OPT}})$ denotes the cost of an optimal offline matching M_i^{OPT} of the first i requests.*

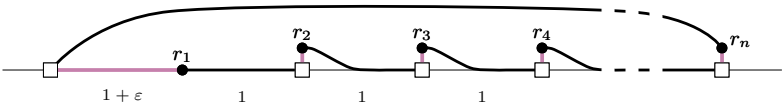
For constructing the online matching M_i' without changing previous matching decisions (no recourse), it may not be possible to augment along a path. Instead, the online t-net-cost algorithm maintains M^* as an auxiliary matching and constructs M_i' by directly connecting the end points r_i and s_i of the augmenting path P_i. That is, $M_i' := M_{i-1}' \cup \{(r_i, s_i)\}$, cf. Figure 1.1a. In particular, M_i' and M_i^* utilize the same sets of servers.

Intuitively, in putting a higher weight on edges that would be added to M^* during augmentation, the parameter t in the t-net cost function discourages the offline t-net-cost algorithm from choosing long paths for augmentation (with respect to actual costs, not in the t-net-cost metric). This allows for a trade-off between minimizing the cost of the underlying offline matching and minimizing the connection costs in M' with the latter occurring in a greedy fashion. Looking at the extremal cases, this becomes even more clear.

When $t = 1$, the offline t-net-cost algorithm is in fact equivalent to the Hungarian Method [KY55] which computes an optimal offline solution. The corresponding online matching, however, is that of the Permutation algorithm [KP93; KMV94] and has a competitive ratio of $\Theta(n)$, see [KP93; KMV94] and the lower bound instance in Figure 1.1a. In contrast, as t tends to infinity, the algorithm prefers augmenting via paths of smaller length and its behavior becomes more similar to that of the greedy online algorithm matching a request upon arrival to the nearest free server. The competitive ratio in this case is in $\Theta(n)$, see [Rag16] and Figure 1.1b. Interestingly though, when $t = 3$, the t-net-cost algorithm has a competitive ratio of $O(\log n)$ [Rag18].



(a) An (alternating) instance where the t-net-cost algorithm, with $t = 1$, has a competitive ratio of $\Omega(n)$. At time i, the optimal offline matching M^* increases in cost by 1, while M' increases in cost by $2i - 1 = c(e_i)$. Thus $\mathrm{OPT} = n$ and $c(M') = \Omega(n^2)$.



(b) An instance where the t-net-cost algorithm, with $t \to \infty$, has a competitive ratio of $\Omega(n)$. For i with $1 \leq i < n$, the augmenting path from r_i to the next free server on the left has a cost of $t \cdot (1+\varepsilon) - (i-1)$ and to the next free server on the right cost t. With $\varepsilon > \frac{1}{t}(i-1) \to 0$, the t-net-cost algorithm augments to the right resulting in a cost of $\Omega(n)$, while $\mathrm{OPT} = 1+\varepsilon$.

Fig. 1.1: Lower-bound instances for the t-net-cost algorithm and extremal choices of $t = 1$ and $t \longrightarrow \infty$.

1.2 A Constant-Competitive Algorithm with Bounded Recourse

In this section, we prove our main results, Theorems 1.1 and 1.2, by describing the corresponding algorithms. We start by giving a high-level overview of our algorithm for Theorem 1.1. It exploits the properties of the t-net-cost algorithm by carefully balancing between the offline matching M^* and the online matching M', simultaneously bounding competitive ratio and recourse budget. On a high level, this is done as follows. When a request arrives, we match it as in M' and locally group it with other recent requests into *blocks* that correspond to intervals on the line. Matching requests as in the online matching increases the total cost but requires zero recourse. A structural result, Lemma 1.13, allows us to bound this increase in cost, but only for a set of blocks whose corresponding intervals are pairwise disjoint. Thus, when new blocks are created, the requests in old intersecting blocks need to be

rematched according to M^* causing a local update, which we call a *recourse step*. During such a recourse step, the changes in M^* caused by the arrivals of the requests in the respective old blocks are applied simultaneously. This eliminates any redundant recourse actions that would have occurred when updating M^* repeatedly, for each arriving request. Intuitively, blocks can therefore be seen as input buffers for M^* that temporarily match requests as in M' and facilitate an efficient update of M^* in large local batches. The underlying structure of the blocks guarantees that recourse steps affect only the corresponding portion of the line. To prevent the recourse steps from causing too many reassignments, we additionally incorporate an edge freezing scheme for low-cost edges that keeps the overall cost low.

1.2.1 A First Try - Balancing between M^* and M' without Freezing

We start our pursuit of a constant competitive algorithm with bounded recourse by exploring the idea of balancing between M^* and M', and show that this first strategy does indeed yield a constant competitive algorithm. While we will see that this strategy uses potentially a lot of recourse, the algorithm we develop here serves as the basis of our final algorithm satisfying Theorem 1.1. The latter simply requires an additional freezing scheme targeting low-cost edges, which we present in the following subsection.

Definitions and Notation

We classify requests according to the structure of intervals that describe where on the line the t-net-cost algorithm has already searched for free servers, following the ideas from Raghvendra [Rag18]. There, somewhat different definitions used, which we discuss in more detail below, but essentially they serve the same purpose. Throughout the chapter, we visualize the real line as a horizontal line from $-\infty$ on the left to ∞ on the right and use according vocabulary, e.g., by saying a server s is left of a request r, or simply $s < r$. We also assume, without loss of generality, that at any point on the line, there is at most one point of $R \cup S$.

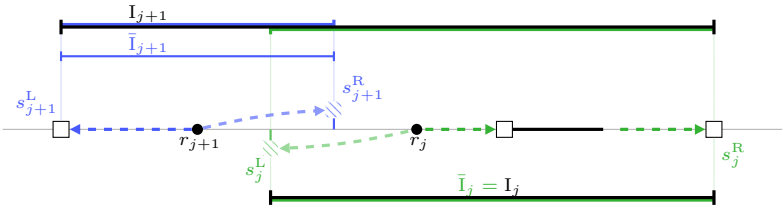


Fig. 1.2: Construction of search intervals and aggregate search intervals for the requests r_j and r_{j+1}. The points s^{L}_j and s^{R}_{j+1} (hatched) are points on the line that do not lie in S. We think of such points as (virtual) servers for the purpose of defining $\bar{\mathrm{I}}_j$ and $\bar{\mathrm{I}}_{j+1}$, respectively.

Define the *search interval* of a request r_i as the open interval $\bar{\mathrm{I}}_i = (s^{\mathrm{L}}_i, s^{\mathrm{R}}_i)$, where s^{L}_i and s^{R}_i denote points on the line farthest to the left and right of r_i, respectively, and reachable from r_i with t-net cost at most $\phi_t^{M^*_{i-1}}(P_i)$. We will show in Lemma 1.10 that one of $s^{\mathrm{L}}_i, s^{\mathrm{R}}_i$ is the server s_i (which r_i is matched to in M'), while the other may not necessarily be a point of $R \cup S$. For the purpose of this definition, we think of it as a (*virtual*) server. That is, we ask the question "If point p was a server in S, would we be able to reach it via an augmenting path of t-net cost at most $\phi_t^{M^*_{i-1}}(P_i)$?". In other words, $\bar{\mathrm{I}}_i$ is the convex hull of all points on the line, reachable from r_i via an augmenting path of t-net cost (strictly) less than $\phi_t^{M^*_{i-1}}(P_i)$.

Define the *aggregate search interval* of r_i as the maximal (open) interval I_i which contains r_i and is a subset of $\bigcup_{j \le i} \bar{\mathrm{I}}_j$. Intuitively, $\bigcup_{j \le i} \bar{\mathrm{I}}_j$ consists of the (disjoint) portions of the line, which the t-net-cost algorithm has already considered, up to time i, in its search for free servers; the interval I_i is simply the connected portion containing r_i. See Figure 1.2 for an illustration. By definition, the portions of the line constituting $\bigcup_{j \le i} \bar{\mathrm{I}}_j$ grow monotonously (and possibly merge while doing so). Therefore, the aggregate search intervals inhibit a laminar structure as is detailed in the following observation.

Observation 1.5. *Whenever $i < j$, then either $\mathrm{I}_i \cap \mathrm{I}_j = \emptyset$ or $\mathrm{I}_i \subseteq \mathrm{I}_j$.*

We say an aggregate search interval I_i is of *level* k if its length satisfies the inequalities $(1+\varepsilon)^{k-1} \le |\mathrm{I}_i| < (1+\varepsilon)^k$; we then write $\ell(\mathrm{I}_i) = k$. Throughout the remainder of this section, we set $t = 3$ and $\varepsilon = \frac{1}{32t}$. Further, two aggregate search intervals are said to belong to the same *block*, if they

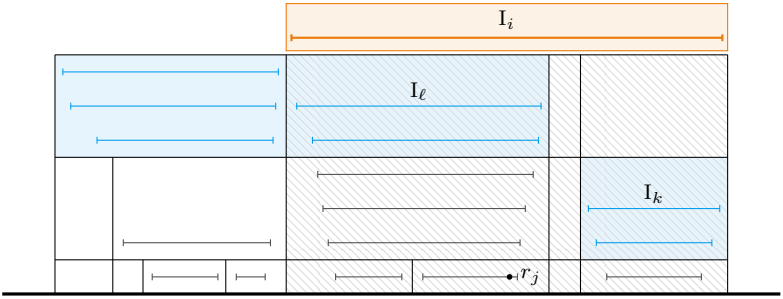


Fig. 1.3: Illustration of a typical block structure. On arrival of r_i, its aggregate search interval I_i creates a new top block (active). Requests in the hatched area are now all inactive. We observe that on I_i, locally, $M^*_{i-1} = M^*_h$ for $\ell, k \leq h < i$.

intersect with each other and are of same level. If the aggregate search intervals of a block are not contained in some interval of a higher level, then this block is said to be a *top block*. We note that the top blocks partition the line into portions that are compatible with the structure of M^*. Specifically, we will see that, at all times, the edges of M^* (and M') are fully contained in some top block. Intuitively, this holds as the portion of the line corresponding to a top block is the convex hull of points considered for the searches for a free server originating from requests inside the block, so no edge of M^* can possibly lead outside the block. A typical block-structure is depicted in Figure 1.3.

Based on these structures, we dynamically classify requests as follows. When the aggregate search interval of a request belongs to a top block then this request is said to be *active*. Once this no longer holds, we call it *inactive*. Note that due to the definition of aggregate search intervals, any arriving request is initially active. And once a request becomes inactive, it will remain inactive indefinitely. When considering the attributes active, inactive, or membership in a block, (or later on *frozen*), we identify a request r_i, the edge $e_i = \{r_i, s_i\} \in M'$ and the interval I_i. For instance, we say r_i is of level k and belongs to a certain block, when this holds for I_i, or, e_i is active when this is the case for r_i.

Algorithm 1.1

Upon arrival of the i-th request r_i: ▷ r_i matched in M' via $e_i = (r_i, s_i)$

Matching r_i

1: match r_i to s_i as in M' ▷ r_i active

Recourse Step

2: **if** there is an active $r_j \in \mathrm{I}_i$ with $\ell(\mathrm{I}_j) < \ell(\mathrm{I}_i)$ **then** ▷ I_i creates top block

3: **for** $r_j \in \mathrm{I}_i$ with $j \neq i$ **do**

4: reassign r_j to the same server as in M^*_{i-1} ▷ r_i inactive

Algorithm Description

Our algorithm, summarized as Algorithm 1.1 produces the *recourse matching* M by assigning requests according to their classification as either active or inactive. Specifically, active requests are matched in M exactly as in the online matching M', and inactive requests are matched as in the offline matching M^*.

While the online matching M' does not change but only gets revealed over time, the offline matching M^* may change its structure drastically at any point causing a lot of recourse. Active requests act as an input buffer for M^*, it is updated only periodically and locally when a new top block appears and renders previously active requests inactive. We call this a *recourse step*.

The algorithm then consists of the following two steps that are executed whenever a new request r_i arrives.

Matching r_i Upon arrival of r_i, label it active and match it according to the online matching M', that is, via the edge $e_i = (r_i, s_i)$.

Recourse Step If there is no $j < i$ such that $\mathrm{I}_j \subseteq \mathrm{I}_i$ and $\ell(\mathrm{I}_j) = \ell(\mathrm{I}_i)$, the arrival of r_i produces a new top block and may render a number of previously active requests inactive, see Figure 1.3. Reassign all requests $r_j \in \mathrm{I}_i$ except r_i to the same server to which they are matched in M^*_{i-1}.

Algorithm Analysis

We start by noting that Algorithm 1.1 indeed produces a matching after each iteration. In Step 1, r_i can be matched to s_i, since both are not yet matched by definition of the t-net-cost algorithm. Therefore, the edge (r_i, s_i) also

does not interfere in a possible recourse step due to the arrival of r_i, as it is not contained in M^*_{i-1}.

Observation 1.6. *At the end of each iteration, the matching M_i computed by Algorithm 1.1 is feasible.*

Bounding the Competitive Ratio We first observe, that the cost of inactive edges can be easily bounded using Lemma 1.4.

Corollary 1.7. *The total cost of inactive edges is at most $t \cdot \mathrm{OPT}$.*

To bound the cost of active edges, we build on the analysis of Raghvendra [Rag18]. We refine his technical propositions and perform a slightly more fine-grained analysis. Instead of simultaneously bounding the cost of all blocks of the same level, we argue more generally on the cost of any set of pairwise disjoint blocks at possibly different levels, where two blocks are said to be *disjoint* if their corresponding maximal aggregate search intervals do not intersect. In particular, we are interested in bounding the cost contribution of the disjoint set of top blocks. We show the following result.

Lemma 1.8. *For $t = 3$, the cost of all active edges of M is bounded by $O(\mathrm{OPT})$.*

Corollary 1.7 and Lemma 1.8 together imply a constant competitive ratio.

Corollary 1.9. *Algorithm 1.1 has a competitive ratio of $O(1)$.*

For the sake of completeness, we give a proof of Lemma 1.8, including a description of our adaptations to [Rag18]. We start by discussing our definition of search intervals. While this definition was motivated by intuition and practicality (specifically with the proof of Lemma 1.19 below in mind), it describes intervals different from the "search intervals" in [Rag18]. However, we show that our definition of aggregate search intervals coincides with Raghvendra's definition of *intervals of a cumulative search region*, which we denote by C_i. We may therefore use the corresponding results from [Rag18].

Lemma 1.10. *For a request r_i, the corresponding aggregate search interval I_i coincides with the "interval of a cumulative search region" C_i defined in [Rag18]. For the search interval $\bar{\mathrm{I}}_i = (s_i^{\mathrm{L}}, s_i^{\mathrm{R}})$, we have $s_i \in \{s_i^{\mathrm{L}}, s_i^{\mathrm{R}}\}$.*

Proof. To see that the first claim holds, note that the intervals C_i and I_i for request r_i are constructed in the same way. That is, they are built by first taking the union of all known search intervals (for the respective definition) which creates a new set of intervals from which we then choose the one that contains r_i. We describe the definition of search intervals in [Rag18], which, to avoid confusion, we call *dual intervals*. The t-net-cost algorithm maintains dual values $y^i : S \cup R \to \mathbb{R}^+$ satisfying $y_s^i + y_r^i = c(s,r)$ if $(s,r) \in M_i^*$ and $y_s^i + y_r^i \leq t \cdot c(s,r)$ otherwise. Additionally, duals of free requests or servers are zero. When a request r_i arrives, a shortest t-net-cost path P_i is found and the duals of all vertices in the search tree, which is partitioned by sets $A_i \subseteq S$ and $B_i \subseteq R$, are updated before augmentation so that the dual constraints on P_i are tight. This is true for both the augmenting paths P_i^{L} and P_i^{R} that are used to reach s_i^{L} and s_i^{R}, respectively. The dual interval of a request r_i is defined as $interior(\bigcup_{r \in B_i} cspan(r,i))$, where $cspan(r,i)$ denotes the interval $[r - \frac{y^i_{\max}(r)}{t}, r + \frac{y^i_{\max}(r)}{t}]$ on the real line and $y^i_{\max}(r)$ the highest dual weight assigned to r until time i.

Raghvendra [Rag16] shows, that the dual constraints on P_i^{L} and P_i^{R} are tight before augmentation. Therefore, we obtain our search intervals $\bar{\mathrm{I}}_i$ from the respective dual interval by replacing $y^i_{\max}(r)$ with the dual weight y_r^i of r before augmentation along P_i. Hence, a search interval is contained in the corresponding dual interval and therefore $\mathrm{I}_i \subseteq C_i$. At the same time, dual weights of requests can only be increased when the request is contained in some B_i or reduced right after an augmentation but remain otherwise unchanged. Thus, the maximal value $y^i_{\max}(r)$ is attained right before an augmentation, in which case the request is part of some P_h^{L} or P_h^{R}. Using again that the dual weights are tight here, this implies $C_i \subseteq \mathrm{I}_i$. Therefore, the intervals I_i and C_i coincide and we may use the respective results from [Rag18].

The second claim follows directly from Lemma 6 in [Rag18], which states that there are no free servers in the dual interval. Intuitively, this holds for our definition of search intervals as well, since a free server inside the

search interval would imply that an augmenting path with strictly lower t-net cost than P_i exists. Formally, one may prove this via induction or use that search intervals are contained in dual intervals, as shown above. Lastly, since clearly $s_i \in [s_i^{\mathrm{L}}, s_i^{\mathrm{R}}]$, it follows that $s_i \in \{s_i^{\mathrm{L}}, s_i^{\mathrm{R}}\}$.

□

We continue with a technical result that allows us to bound the cost contribution of edges of M' whose length is large when compared to the t-net cost of the corresponding augmenting path. Formally, we call an edge e_i of the online matching M' or the corresponding augmenting path P_i *short*, if $c(P_i) \leq \frac{4}{t-1}\phi_t^{M^*_{i-1}}(P_i)$ and *long* otherwise. Raghvendra bounds the cost of all long edges by that of all short ones. We give the following more general statement that allows us to consider a consecutive set of edges instead. Note that the additional additive term of $2(t+1) \cdot \mathrm{OPT}_{j-1}$ when compared to the original statement in [Rag18] disappears again when considering the entire matching, as then $\mathrm{OPT}_{j-1} = \mathrm{OPT}_0 = 0$.

Lemma 1.11. *Let $P_j, P_{j+1}, \ldots, P_k$ be consecutive augmenting paths that are used in the offline t-net-cost algorithm and $e_j, e_{j+1}, \ldots, e_k$ the corresponding edges in M'. Then*

$$\sum_{i:\, e_i \text{ long}} c(e_i) \leq \left(4 + \tfrac{4}{t-1}\right) \sum_{i:\, e_i \text{ short}} c(e_i) \;+\; 2(t+1) \cdot \mathrm{OPT}_{j-1}.$$

Proof. To prove this lemma, we adapt the approach of [Rag18, Lemma 2] for our more general setting. Consider two consecutive matchings M^*_{i-1} and M^*_i, and recall from Equation (1.1) that the t-net cost of the corresponding augmenting path P_i is given by $\phi_t^{M^*_{i-1}}(P_i) = t \cdot c(P_i \setminus M^*_{i-1}) - c(P_i \cap M^*_{i-1})$. The difference in cost of the two matchings can then be expressed as

$$\begin{aligned}
&\tfrac{t+1}{2}\left(c(M^*_i) - c(M^*_{i-1})\right) = \tfrac{t+1}{2}\left(c(P_i \setminus M^*_{i-1}) - c(P_i \cap M^*_{i-1})\right) \\
&\quad = \phi_t^{M^*_{i-1}}(P_i) - (t-1)c(P_i \setminus M^*_{i-1}) + \tfrac{t-1}{2}\left(c(P_i \setminus M^*_{i-1}) - c(P_i \cap M^*_{i-1})\right) \\
&\quad = \phi_t^{M^*_{i-1}}(P_i) - \tfrac{t-1}{2}\left(c(P_i \setminus M^*_{i-1}) + c(P_i \cap M^*_{i-1})\right) \\
&\quad = \phi_t^{M^*_{i-1}}(P_i) - \tfrac{t-1}{2}c(P_i).
\end{aligned}$$

Summing from j to k, and using the fact that $\mathrm{OPT}_{j-1} \le \mathrm{OPT}_k \le c(M_k^*)$ as well as $c(M_{j-1}^*) \le t \cdot \mathrm{OPT}_{j-1}$, we get

$$\sum_{i=j}^{k} \left(\phi_t^{M_{i-1}^*}(P_i) - \tfrac{t-1}{2} c(P_i)\right) = \tfrac{t+1}{2} \sum_{i=j}^{k} \left(c(M_i^*) - c(M_{i-1}^*)\right) \tag{1.2}$$

$$\ge \tfrac{1-t^2}{2} \cdot \mathrm{OPT}_{j-1}. \tag{1.3}$$

Denote by L and H the sets of augmenting paths with index between j and k that are long and short, respectively. Then, Equation (1.2) can be rewritten as

$$\sum_{P_i \in H} \phi_t^{M_{i-1}^*}(P_i) \tag{1.4}$$

$$\overset{(1.2)}{\ge} \tfrac{t-1}{2} \sum_{P_i \in H} c(P_i) + \sum_{P_i \in L} \left(\tfrac{t-1}{2} c(P_i) - \phi_t^{M_{i-1}^*}(P_i)\right) + \tfrac{1-t^2}{2} \cdot \mathrm{OPT}_{j-1}$$

$$\ge \sum_{P_i \in L} \phi_t^{M_{i-1}^*}(P_i) + \tfrac{1-t^2}{2} \cdot \mathrm{OPT}_{j-1}, \tag{1.5}$$

where the last inequality uses the fact that $\sum_{P_i \in H} c(P_i)$ is non-negative and that for a long path P_i we have $c(P_i) \ge \frac{4}{t-1} \cdot \phi_t^{M_{i-1}^*}(P_i)$. Using again non-negativity of $\sum_{P_i \in H} c(P_i)$, as well as Equations (1.2) and (1.5), we get

$$2 \sum_{P_i \in H} \phi_t^{M_{i-1}^*}(P_i) \overset{(1.5)}{\ge} \sum_{P_i \in H \cup L} \phi_t^{M_{i-1}^*}(P_i) + \tfrac{1-t^2}{2} \cdot \mathrm{OPT}_{j-1} \tag{1.6}$$

$$\overset{(1.2)}{\ge} \tfrac{t-1}{2} \sum_{P_i \in H \cup L} c(P_i) + (1-t^2) \cdot \mathrm{OPT}_{j-1} \tag{1.7}$$

$$\overset{(1.2)}{\ge} \tfrac{t-1}{2} \sum_{P_i \in L} c(P_i) + (1-t^2) \cdot \mathrm{OPT}_{j-1}. \tag{1.8}$$

By definition of t-net cost and using the triangle equality, we obtain the inequalities $t \cdot c(e_i) \ge \phi_t^{M_{i-1}^*}(P_i)$ and $c(P_i) \ge c(e_i)$, respectively. After dividing Equation (1.6) by $\frac{t-1}{2}$, this implies the lemma statement

$$\tfrac{4t}{t-1} \sum_{P_i \in H} c(e_i) \ge \sum_{P_i \in L} c(e_i) - 2(t+1) \cdot \mathrm{OPT}_{j-1}.$$

□

Consider a level-k block B and denote by $\mathcal{B}$ the instance that consists of the requests and servers that are the endpoints of edges in M' belonging to B. Further denote by $M'_{\mathcal{B}}$ the set of these edges. We aim to bound the cost of $M'_{\mathcal{B}}$ for several disjoint blocks simultaneously using the previous lemma.

To be able to do this, we first need to argue that it is okay to assume, for the sake of cost analysis, that the requests arrive in a certain order. In particular, we argue that as long as the relative order of requests with overlapping aggregate search intervals is preserved, the structure and cost of both the offline and the online matching somewhat remains unchanged. Specifically, when a request arrives, in any of such arrival orders satisfying the above, the offline and online matching are identical on the portion of the line corresponding to the requests aggregate search interval. In particular, when the n-th request arrives, the final matchings are identical.

Lemma 1.12. *Let $\pi \in S_n$ be a permutation that is such that $i < j$ and $\mathrm{I}_i \subseteq \mathrm{I}_j$ imply $\pi(i) < \pi(j)$. Changing the arrival order of requests to $r_{\pi(1)}, \ldots, r_{\pi(n)}$ does not change the final matchings M' and M^*. Specifically, after the change,*

(i) if r_i and r_j belonged to block B, then so do $r_{\pi(i)}$ and $r_{\pi(j)}$ now, and
(ii) the cost of $M'_{\mathcal{B}}$ is unchanged compared to the original arrival order.

Proof. By definition of search intervals, only requests and servers inside the closure of $\bar{\mathrm{I}}_i$ are considered by the t-net-cost algorithm when request r_i arrives, and are thus the only ones possibly affected by augmentation. Therefore, we can assume by induction, that right before the arrival of r_i, the online and offline matching are identical within $\bar{\mathrm{I}}_i$ for any permutation π that satisfies the above requirements. The t-net-cost algorithm then uses the same path P_i for augmentation, irrespective of π, which implies the lemma statement. □

We are now ready to bound the cost of M' for a set of disjoint blocks.

Lemma 1.13. *For disjoint blocks $B_1, B_2, \ldots, B_h$, and with $t = 3$, we have*

$$\sum_{i=1}^{h} c(M'_{\mathcal{B}_i}) \leq 5600 \cdot \mathrm{OPT}.$$

Proof. Since we consider a set of disjoint blocks, we may use Lemma 1.12 to change the arrival order of requests and assume that those requests that belong to the blocks $B_1, B_2, \ldots, B_h$ arrive consecutively and that no requests on other portions of the line have arrived yet. Denote by M_L and M_H the set of long and short edges of $M'_{\mathcal{B}_1} \cup M'_{\mathcal{B}_2} \cup \ldots \cup M'_{\mathcal{B}_h}$, respectively, and by $\mathrm{OPT}_{\mathcal{B}_i}$ the cost of an optimal matching on $\mathcal{B}_i$. Together, Lemmas 3 and 13 from [Rag18] imply

$$c(M_H) \leq 99 \cdot \sum_{i=1}^{h} \mathrm{OPT}_{\mathcal{B}_i} + \tfrac{2}{47} \cdot c(M_L).$$

We then use Lemma 12 from [Rag18] to obtain

$$c(M_H) \leq 594 \cdot \mathrm{OPT} + \tfrac{2}{47} \cdot c(M_L).$$

Note that the lemma is only stated for a set of level-k blocks, but the proof carries over verbatim for the case of arbitrary disjoint blocks. After bounding $c(M_L)$ with the use of Lemma 1.11, we get

$$c(M_H) \leq 594 \cdot \mathrm{OPT} + \tfrac{2}{47}(6 \cdot c(M_H) + 8 \cdot \mathrm{OPT}),$$

which in turn implies $c(M_H) \leq 798.2 \cdot \mathrm{OPT}$. Using one last time Lemma 1.11, we conclude that

$$\sum_{i=1}^{h} c(M'_{\mathcal{B}_i}) = c(M_H) + c(M_L) \leq 7 \cdot c(M_H) + 8 \cdot \mathrm{OPT} \leq 5600 \cdot \mathrm{OPT}.$$

□

Proof of Lemma 1.8. To bound the cost of all active edges, recall that active edges are precisely those edges that are contained in a top block, and further, that at any given time any two top blocks are disjoint. Thus, we may apply Lemma 1.13 to the top blocks of M. This directly implies the statement of Lemma 1.8, concluding the competitive analysis of Algorithm 1.2. □

And the Recourse Budget? With the majority of requests matched as in the *offline* matching M^*, it should come as no surprise that Algorithm 1.1 performs poorly in terms of recourse budget. To see this, consider the example

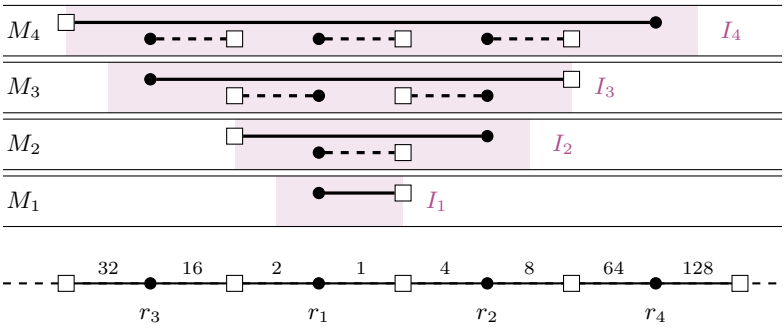


Fig. 1.4: An alternating instance with exponentially increasing connection costs. Edge costs are indicated above the corresponding portion of the line (drawing not to scale). Active edges are drawn solid, inactive edges dashed. Note that the aggregate search intervals (gray) grow exponentially in size. Therefore, a (top) block consists of a single aggregate search interval only and every arrival of a request triggers a recourse step, reassigning all existing inactive requests. This underlines the need for a freezing scheme.

depicted in Figure 1.4 showing an alternating instance with exponentially increasing edge costs. At the arrival of a request r_i with $i > 2$, all requests r_j with $j < i$ are reassigned. Thus, the recourse budget is linear and Algorithm 1.1 does not satisfy the requirements of Theorem 1.1. However, our efforts on Algorithm 1.1 are not in vain, since its shortcomings in terms of recourse budget can be remedied with a simple extension described in the following subsection.

1.2.2 Adding a Freezing Scheme to Obtain Constant Recourse

Consider again Figure 1.4. The exponentially increasing edge costs that thwarted our success with Algorithm 1.1, turn out to also be the key to salvaging it. Intuitively, if edge costs do not increase too much, then a top block can hold more requests, so there will not be as many requests of different levels. As this translates directly into recourse actions, everything is already fine with Algorithm 1.1. If on the other hand edge costs do increase a lot, say exponentially as in Figure 1.4, then at some point, the cost-contribution of small edges becomes negligible. Spending recourse actions on such edges would be absurd, so we instead choose to keep them as they are, we *freeze*

them. We would still like to match other inactive requests that are not frozen with respect to M^*. However, since a frozen edge may block a server that in M^* is assigned to a different request r, we are forced to make a compromise and match r to a different, farther-away server via a *detour* taken along the frozen edge.

The Detour Matching We describe a way of combining two matchings that initially seem incompatible, e.g., because some request is matched to different servers in the two matchings. Let $\tilde{M}, M$ be matchings such that $\tilde{M} \oplus M$ consists of alternating paths that are augmenting with respect to M. Such a path starts in a request and ends in a server, both of which are unmatched in M. We call the matching that consists of precisely these request-server pairs the *detour matching* of $\tilde{M}$ with respect to M. When clear from context, we simply refer to this matching as $\tilde{M} \oplus M$. For an illustration, see Figure 1.5. For an example, consider the online t-net-cost algorithm. The symmetric difference of M^*_{i-1} and M^*_i consists of one path that is augmenting with respect to M^*_{i-1}, namely P_i. The detour matching of M^*_i with respect to M^*_{i-1} then consists of precisely the edge $e_i \in M'_i$, the *detour* that one takes when not augmenting along P_i but connecting the ends of the path directly. In metric spaces, it is easy to see via the triangle inequality, that the cost of the detour matching is bounded by the sum of costs of the two generating matchings.

Observation 1.14. *For matchings $\tilde{M}, M$ whose symmetric difference consists of alternating paths that are augmenting with respect to M, the detour matching $\tilde{M} \oplus M$ satisfies $c(\tilde{M} \oplus M) \leq c(\tilde{M}) + c(M)$.*

A Simple Freezing Scheme Denote by $F \subseteq M'$ the set of frozen edges and consider the following simple freezing scheme for which we need to make the assumption that the precise values of OPT and n are known a priori. Freeze all edges of M' with cost $\frac{\mathrm{OPT}}{n}$ or less and match inactive requests with respect to the detour matching $M^* \oplus F$. Since there are at most n edges in F, the cost of frozen edges is bounded by OPT. Therefore, using Observation 1.14, we conclude that the cost of inactive edges is bounded by $c(F) + c(M^* \oplus F) \leq (t+2) \cdot \mathrm{OPT}$. The cost of active edges remains

in $O(\mathrm{OPT})$ since nothing changed from Section 1.2.1. One can further show, that due to this freezing scheme, any request participates in only $O(\log n)$ recourse steps. The proof follows the intuition described above: If a request r would take part in $o(\log n)$ recourse steps, then it is contained in $o(\log n)$ intervals of increasing level. However, edge costs are increasing exponentially and r must in fact already be frozen.

While we would like to follow this general strategy as well, in the online model, we do not know OPT or n. Therefore, we need a dynamic freezing scheme, targeting for instance $\frac{\mathrm{OPT}_i}{i}$ as a threshold for freezing in round i. A typical guess-and-double approach may work concerning the costs. Yet, care has to be taken as $\frac{\mathrm{OPT}_i}{i}$ is not monotone. A major obstacle appears to be bounding the recourse budget. The details of the algorithm and dynamic freezing scheme as well as their analysis are given in the remainder of this subsection.

Algorithm Description

Since the difference to Algorithm 1.1 is mainly the added freezing scheme, we start with its description. At time i, we say an inactive edge $e_j \in M'$ is frozen if its cost is below $\frac{\mathrm{OPT}_i}{i^2}$, and denote by F_i the set of all such edges. Furthermore, if for a previously frozen e_j we have that $c(e_j) > \frac{\mathrm{OPT}_i}{i}$, then we *unfreeze* the edge and delete it from F_i. Note that we use the edge e_j of the online matching to determine whether r_j is frozen, but r_j is possibly matched in M_i via a different edge.

While we generally aim to match frozen requests as in the online matching M', this assignment is not implemented until the request takes part in a recourse step. We show that this prevents unnecessary recourse actions that could be caused by requests repeatedly alternating between being frozen and unfrozen. Denote by $M_i^{\mathrm{F}} \subseteq F_i$ the subset of frozen edges that is in M_i, i.e., the set of edges where the aforementioned assignment according to M' was implemented during a recourse step. When unfreezing a request, however, there is no delay and the corresponding change must be reflected in the matching immediately, since the cost contribution of the frozen edge is now too high.

Algorithm 1.2

Upon arrival of the i-th request r_i: ▷ r_i matched in M' via $e_i = (r_i, s_i)$

Matching r_i

match r_i to s_i as in M' ▷ label r_i active

Freezing/Unfreezing

determine the set F_i of frozen edges

for inactive requests r_j that become unfrozen **do**

  remove e_j from M_i^{F}

  repair assignments on corresponding path of $M^*_{h(i,j)} \oplus M_i^{\mathrm{F}}$

Recourse Step

if there is an active $r_j \in \mathrm{I}_i$ with $\ell(\mathrm{I}_j) < \ell(\mathrm{I}_i)$ **then** ▷ I_i creates top block

  for $r_j \in \mathrm{I}_i$ recently frozen **do** ▷ label $r_j \in \mathrm{I}_i \setminus \{r_i\}$ inactive

    add $e_j = (r_j, s_j)$ to M_i^{F} and reassign r_j to s_j

  for unfrozen $r_j \in \mathrm{I}_i$ with $j \neq i$ **do**

    reassign r_j according to $M^*_{i-1} \oplus M_i^{\mathrm{F}}$

With the addition of the freezing scheme, our algorithm, summarized as Algorithm 1.2, now partitions the requests into three groups: (i) active requests matched according to the online matching M', (ii) inactive, frozen requests in M^{F}, also matched according to M', and (iii) inactive requests that are not in M^{F}, matched according to the detour matching $M^* \oplus M^{\mathrm{F}}$. Similarly, the algorithm itself is now divided into three steps, that are all executed when a new request arrives:

Matching r_i Upon arrival of request r_i, label it active and match it according to the online matching M', that is, via the edge $e_i = (r_i, s_i)$.

Freezing/Unfreezing Update the set F_i of frozen edges according to the following freezing rules: Freeze an edge $e_j \in M'$ if its cost is below $\frac{\mathrm{OPT}_i}{i^2}$, and unfreeze a previously frozen edge $e_j \in M'$ if $c(e_j) > \frac{\mathrm{OPT}_i}{i}$. Delete unfrozen edges from F_i and M_i^{F}, and reassign them with respect to the detour matching $M^*_{h(i,j)} \oplus M_i^{\mathrm{F}}$, where $h(i,j)$ is defined as the index of the highest-level inactive interval at time i which contains r_j. That is, we have $h(i,j) = \max\{k \leq i \mid r_j \in \mathrm{I}_k \text{ and } \mathrm{I}_k \text{ is not in a top block}\}$. For an illustration of $h(i,j)$, see Figure 1.3, where $h(i,j) = \ell$. Reassignments due to unfreezing are efficiently implemented as follows.

Consider an edge $e = (s, r)$ right before it is unfrozen. If e is already matched according to the offline matching, then there is nothing to do. Otherwise, it must be part of an alternating path or cycle in $M^* \oplus M^{\mathrm{F}}$. In the latter case,

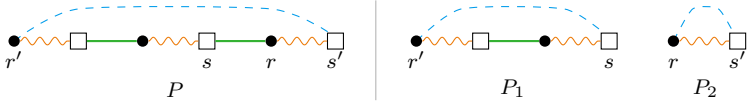


Fig. 1.5: Illustration of Step 4 in Algorithm 1.2. Edges of M^* are drawn wiggled, edges of M^{F} solid and edges of the detour matching $M^* \oplus M^{\mathrm{F}}$ dashed. After unfreezing r, the removal of (r, s) splits the path P in $M^* \oplus M^{\mathrm{F}}$ in two augmenting paths P_1 and P_2.

again, no recourse action needs to be taken as the removal of e from M^{F} results in a path from r to s. In the detour matching, we want to connect the ends of this path, which is already accomplished by the edge $e = (s, r)$. Consider the case that $e \in P$ for some alternating path P in $M^* \oplus M^{\mathrm{F}}$ that starts in a request r' and ends in a server s'. Unfreezing e and matching according to $M^* \oplus M^{\mathrm{F}}$ decomposes P into the r'-s-path P_1 and the r-s'-path P_2. In Algorithm 1, we implement these changes via two recourse actions: we reassign r to s' and r' to s; see Figure 1.5.

Recourse Step If there is no $j < i$ such that $\mathrm{I}_j \subseteq \mathrm{I}_i$ and $\ell(\mathrm{I}_j) = \ell(\mathrm{I}_i)$, the arrival of r_i produces a new top block and triggers a recourse step. In this case, we assign requests in I_i that were recently frozen according to M' and add the corresponding edge to M_i^{F}. Next, we reassign all other requests (unfrozen, inactive) that lie in I_i according to $M_{i-1}^* \oplus M_i^{\mathrm{F}}$, the detour matching.

Algorithm Analysis

We start again by verifying that Algorithm 1.2 produces a matching after each iteration. As before, r_i can be matched to s_i in Step 1. Similarly, active edges do not interfere in the unfreezing procedure (Step 4) due to the definition of $h(i, j)$. It remains to argue that we can indeed construct the detour matching as described. To this end, we consider the symmetric difference $M^* \oplus M^{\mathrm{F}}$ and show that, for an unfrozen request r, there is an augmenting path P from r to a server s' not covered by M^{F}. To see this, note that in $M^* \oplus M^{\mathrm{F}}$ the unfrozen r is incident only to an edge $(r, s) \in M^*$, so it cannot be contained in a cycle. Additionally, the path starting with (r, s) cannot end at a request, since the remaining requests on the path are covered by an edge of M^{F} and M^* each. Thus, it must end at a server s' that is not matched in M^{F}, as required.

Observation 1.15. *At the end of each iteration, the matching M_i produced by Algorithm 1.2 is feasible.*

When bounding the competitive ratio, we may use the results from Section 1.2.1. By Lemma 1.8, the cost of active edges is in $O(\mathrm{OPT})$. Using Observation 1.14 and the definition of the freezing scheme, we cab bound the cost of inactive edges by $c(M^{\mathrm{F}}) + c(M^* \oplus M^{\mathrm{F}}) \leq (t+2) \cdot \mathrm{OPT}$. Together, this implies that the competitive ratio of Algorithm 1.2 is constant.

Corollary 1.16. *Algorithm 1.2 has a competitive ratio of $O(1)$.*

Bounding the Recourse Due to delaying reassignments until an appropriate recourse step, freezing a request r does not cause any immediate recourse actions. Unfreezing r, on the other hand, may cause reassignments (Step 4), namely of r and possibly one additional request, see Figure 1.5. We charge these two recourse actions to r, in particular, to the last recourse step r participated in. However, recourse due to unfreezing r happens at most once between two consecutive recourse steps of r. Thus, a request is charged at most three times per recourse step it participates in, once for the recourse step itself and possible two times more for a subsequent unfreezing.

Observation 1.17. *A request is charged at most three times the number of recourse steps it is involved in.*

For a request that was frozen at time i and unfrozen at time $j > i$, we know that $c(e) \leq \frac{\mathrm{OPT}_i}{i^2}$ and $c(e) > \frac{\mathrm{OPT}_j}{j}$ for the corresponding edge $e \in M'$. Since $\mathrm{OPT}_i \leq \mathrm{OPT}_j$, this implies the following.

Observation 1.18. *A request frozen at time i stays frozen at least until time i^2.*

On the other hand, the number of recourse steps in which a continuously unfrozen request takes part in can be bounded from above.

Lemma 1.19. *If a request $r \in \mathrm{I}_i \subseteq \mathrm{I}_j$ is not frozen from time i to time j, then it holds that $\ell(\mathrm{I}_j) - \ell(\mathrm{I}_i) = O(\log j)$. In particular, between time i and j, there are $O(\log j)$ recourse steps in which r can participate.*

Proof. Consider the search interval $\bar{\mathrm{I}}_j = (s_j^{\mathrm{L}}, s_j^{\mathrm{R}})$. By definition of search intervals, there exist augmenting paths $P_j^{\mathrm{L}}, P_j^{\mathrm{R}}$ connecting r_j to s_j^{L} and s_j^{R} with the same t-net cost as P_j, the path used by the t-net-cost algorithm. We bound the length of $P_j \in \{P_j^{\mathrm{L}}, P_j^{\mathrm{R}}\}$ by

$$c(P_j) = c(P_j \cap M_{j-1}^*) + c(P_j \cap M_j^*) \leq 2t \cdot \mathrm{OPT}_j. \tag{1.9}$$

Without loss of generality, assume $P_j = P_j^{\mathrm{L}}$. Interpreting the point s_j^{R} as a virtual server and assuming that it is contained in S, we could augment M_{j-1}^* also along P_j^{R} yielding a different matching $\tilde{M}_j^*$. By definition of P_j^{R} and Equation (1.1), we have

$$\begin{aligned} t \cdot c(P_j \cap M_j^*) - c(P_j \cap M_{j-1}^*) &= \phi_t^{M_{j-1}^*}(P_j) = \phi_t^{M_{j-1}^*}(P_j^{\mathrm{R}}) \\ &= t \cdot c(P_j^{\mathrm{R}} \cap \tilde{M}_j^*) - c(P_j^{\mathrm{R}} \cap M_{j-1}^*). \end{aligned}$$

Thus,

$$\begin{aligned} c(P_j^{\mathrm{R}}) &= c(P_j^{\mathrm{R}} \cap M_{j-1}^*) + c(P_j^{\mathrm{R}} \cap \tilde{M}_j^*) \\ &\leq c(P_j^{\mathrm{R}} \cap M_{j-1}^*) + c(P_j \cap M_j^*) + \frac{1}{t} \cdot c(P_j^{\mathrm{R}} \cap M_{j-1}^*) \\ &\leq 3t \cdot \mathrm{OPT}_j. \end{aligned}$$

From Equation (1.9), we obtain $|\bar{\mathrm{I}}_j| \leq c(P_j^{\mathrm{L}}) + c(P_j^{\mathrm{R}}) \leq 5t \cdot \mathrm{OPT}_j$,. This implies that $|\mathrm{I}_j| \leq \sum_{k \leq j} |\bar{\mathrm{I}}_k| \leq 5t \cdot j \cdot \mathrm{OPT}_j$. On the other hand, the cost of $|\mathrm{I}_i|$ can be bounded from below by $c(e)$, where e is the edge in M' incident to r, as both endpoints of e are contained in the interval. We then obtain

$$\frac{|\mathrm{I}_j|}{|\mathrm{I}_i|} \leq \frac{5t \cdot j \cdot \mathrm{OPT}_j}{c(e)} < 5t \cdot j^3. \tag{1.10}$$

The last inequality follows from the assumption that r is not frozen at time j, which implies $c(e) > \frac{\mathrm{OPT}_j}{j^2}$. Recall that, by the definition of levels, we have $|\mathrm{I}_i| < (1+\varepsilon)^{\ell(\mathrm{I}_i)}$ and $(1+\varepsilon)^{\ell(\mathrm{I}_j)-1} \leq |\mathrm{I}_j|$. Thus, Equation (1.10) allows us to conclude

$$\ell(\mathrm{I}_j) - \ell(\mathrm{I}_i) \leq 1 + \log_{(1+\varepsilon)}(5t \cdot j^3) = 1 + \log_{(1+\varepsilon)}(5t) + \frac{3 \log j}{\log(1+\varepsilon)} \leq c \cdot \log j\,, \tag{1.11}$$

for some constant c. Regarding the second claim, recall that a request r participates in a recourse step whenever an interval containing r opens a new top block (i.e., a new level) while r is not (freshly) frozen. Between

time i and time j, this can only happen for intervals I_h of distinct levels for which $\mathrm{I}_i \subseteq \mathrm{I}_h \subseteq \mathrm{I}_j$. The claim then directly follows Equation (1.11). □

We can now use Lemma 1.19 and Observation 1.18 to bound the total number of recourse actions taken by Algorithm 1.2.

Lemma 1.20. *Algorithm 1.2 uses a recourse budget of at most* $O(\log n)$.

Proof. Consider a request r. By Observation 1.17, it suffices to bound the number of recourse steps that r is involved in. Let $[i_h^U, i_h^F]$, for $h = 0, 1, \ldots, k$, be maximal intervals of consecutive time points during which r is not frozen, i.e., r is not frozen only at the time points i with $i \in [i_h^U, i_h^F]$ for some h. We use induction on k to show that r participates in at most $2c \cdot \log(i_k^F)$ recourse steps, where c is the constant from Equation (1.11). The base case, $k = 0$, follows directly from Lemma 1.19. For $k \geq 1$, we use Observation 1.18 to obtain $(i_{k-1}^F)^2 \leq i_k^U \leq i_k^F$. By induction hypothesis, the number of reassignments that involve r in the first $k-1$ time intervals is at most

$$2c \cdot \log(i_{k-1}^F) \leq 2c \cdot \log\left(\sqrt{i_k^F}\right) = c \cdot \log(i_k^F).$$

For the last time interval, we have at most $c \cdot \log(i_k^F)$ such recourse steps by Lemma 1.19. Since $i_k^F \leq n$, this concludes the proof. □

Proof of Theorem 1.1. Corollary 1.16 and Lemma 1.20 directly imply the validity of Theorem 1.1. This concludes the analysis of Algorithm 1.2. □

1.2.3 A Scalable Algorithm

In the previous subsection, we presented an algorithm, Algorithm 1.2, that has a constant competitive ratio at the price of a recourse budget that is bounded by $O(\log n)$. While the focus was on the former, i.e., on achieving a constant competitive ratio, it may be of practical interest to more flexibly balance the cost performance of the algorithm with its recourse budget instead, as described in Theorem 1.2. Indeed, all the building blocks needed for the proof of this theorem have already been established in the previous

subsections. As we now show, a slight alteration to Algorithm 1.2 already yields the desired cost-recourse trade-off.

Theorem 1.2. *For any function $f : \mathbb{N} \longrightarrow \mathbb{N}$ with $1 \leq f(n) \leq \log n$, there is an algorithm for online minimum-cost matching on the line that is $O(f(n))$-competitive and has an amortized recourse budget of $O(\frac{\log n}{f(n)})$.*

Proof. To obtain an algorithm that is $O(f(n))$-competitive and uses a recourse budget of $O(\frac{\log n}{f(n)})$, we alter Algorithm 1.2 as follows. Intuitively, and in the terms of the high-level overview from the beginning of this section, we want the input buffers (top blocks) to have a higher capacity (more active requests), leading to a larger cost due to the contribution of edges in M' but lower recourse due to less frequent updates. We achieve this by "vertically" (consider the visualization in Figure 1.3) enlarging the area of top blocks and allowing the requests of several non-disjoint blocks to be active. However, intersecting active blocks may not differ in their level by more than $f(n)$, which specifies the "height" of the input buffer.

Specifically, all we do is change Step 5 of Algorithm 1.2 as follows.

5: **if** there is an active $r_j \in \mathrm{I}_i$ such that $\ell(\mathrm{I}_j) \leq \ell(\mathrm{I}_i) - f(n)$ **then**

It is easy to see, that the cost of active edges is then in $O(f(n) \cdot \mathrm{OPT})$, since the set of blocks with active requests can be partitioned into at most $f(n)$ sets of disjoint blocks to which our original analysis can be applied separately.

Similarly, when considering the recourse budget, note that the first part of the statement of Lemma 1.19 remains true, but now implies that an unfrozen request can participate in at most $O(\frac{\log j}{f(n)})$ recourse steps. Lemma 1.20 can easily be adapted to account for the additional factor of $\frac{1}{f(n)}$ which concludes the proof of the theorem. □

To see that our analysis is asymptotically optimal, we consider the family of lower bound instances (for the online setting) described in [AFT18]. Roughly, they are constructed as follows.

An instance of the family consists of $n = 2^h - 1$ requests and $n+1$ servers that alternate on the line. That is, requests are placed at points $2i-1$ for $i \in [n]$ and servers at points $2i-2$ for $i \in [n+1]$, where $[n]$ denotes the set $\{1, 2, \ldots, n\}$.

The requests appear in h batches, each of which contains every second request of those that have not appeared yet: Initially, requests $4i-3$ for $i \in [2^{h-2}]$, then requests $8i-5$ for $i \in [2^{h-3}]$, and so on. Restricting the class of considered algorithms by some assumptions, which are satisfied for all known deterministic online algorithms to date, one can show the following properties. Any such algorithm matches a requests of batch i at minimum cost to either the nearest free server to the left or to the right. When partitioning requests of a batch into pairs of consecutive requests, the requests of a pair match towards each other, i.e., to the two free servers between the two requests. Between those two servers, in turn, always lies a single request which is contained in the next batch. Thus, the cost of an edge in batch i is precisely 2^{i-1}, which shows the lower bound. For details, see [AFT18].

Structurally, the exponentially increasing edge costs mean that every arriving request creates a new top block, similarly as in Figure 1.4. Additionally, the structure of aggregate search intervals resembles a full binary tree. Specifically, there are $\log n$ levels and all $\frac{n+1}{2}$ requests on the lowest level will participate in $\frac{\log n}{f(n)}$ recourse steps due to the arrival of their higher level ancestors. At the same time, no request is frozen, as $\mathrm{OPT}_i = i$ and the minimum cost of an edge is 1. Therefore, we have a lower bound of $\Omega(\frac{\log n}{f(n)})$ for the recourse budget of our algorithm.

1.3 A Near-Optimal Algorithm on Alternating Instances

In this section, we consider alternating instances, i.e., instances on which requests and servers alternate on the line. We assume without loss of generality that $|S| = |R|$. For such instances, an optimum (offline) matching assigns all requests either to the server directly to their left or all requests to the server on their right. Denote these matchings by M^{L} and M^{R}, respectively, and call their edges *minimal*.

We describe a $(1+\varepsilon)$-competitive algorithm for alternating instances that reassigns each request at most a constant number of times. In addition to its output M, it maintains M^*, the matching produced by the offline t-net-cost algorithm, and a set of frozen edges M^{F}. In contrast to Algorithm 1.2, we are not only able to disregard M', but furthermore we can employ a

much simpler freezing scheme. Specifically, a request is frozen when it is reassigned for the k-th time, for some k only depending on ε. The request remains matched to its current server perpetually and the corresponding edge is added to M^{F}. Non-frozen requests are matched according to the detour matching $M^* \oplus M^{\mathrm{F}}$ as described in Section 1.2. By design, the recourse budget per request is constant, only the competitive analysis remains. The algorithm is summarized as Algorithm 1.3.

Observe that when considered as line segments, the augmenting paths P_i used by the t-net-cost algorithm have a laminar structure on alternating instances, since after time i, there are no free requests or servers in the portion of the line corresponding to P_i. This allows us to view augmenting paths as nodes of a forest, the *path forest*, where P_i is a child of the minimal augmenting path that properly contains it, or a root if no such path exists (see Figure 1.6). The *depth* of a path P_i denotes its distance to the root of its tree and determines the number of reassignments of the corresponding request r_i in M^*.

The intuition why such an extremely simple freezing scheme as utilized in Algorithm 1.3 works on alternating instances, relies on the fact that, here, we can draw a direct connection between the length of a path P_i and its depth in the path forest. Specifically, we will show in Lemma 1.23, that the lengths of augmenting paths grow exponentially when moving from a leaf of the path forest to the corresponding root. Since ancestors of a request correspond to reassignments in M^*, we can charge the cost of frozen edges of a certain depth as well as possible detours to higher level ancestors. The per-request recourse budget is thus directly connected to the competitive

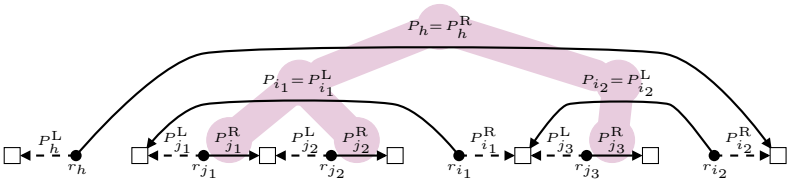


Fig. 1.6: Illustration of a path tree (red) in an alternating instance. Paths not chosen for augmentation are dashed. Servers are depicted as squares and requests as filled circles.

Algorithm 1.3 with per-request recourse budget $k \in \mathbb{N}$

initialize recourse counter $h_r \leftarrow 0$ for all $r \in R$, $M^{\mathrm{F}} \leftarrow \emptyset$

Upon arrival of the i-th request r_i:

1: update M^*_{i-1} by augmenting along P_i, and set $h_r \leftarrow h_r + 1$, for all $r \in P_i$
2: **for all** $r \in P_i$ with $h_r > k$ **do** ▷ r frozen
3: $M^{\mathrm{F}} \leftarrow M^{\mathrm{F}} \cup \{(r, s)\}$, where s is the server currently matched to r
4: **for all** $r \in P_i$ with $h_r \leq k$ **do** ▷ r not frozen
5: reassign r with respect to $M^*_i \oplus M^{\mathrm{F}}$

ratio since a larger recourse budget means charging to a higher ancestor with exponentially larger cost.

Notation We use a similar interval structure as before and keep the same notation. The definition of the intervals is slightly changed: Consider intervals $\mathrm{I}_i = [s_i^{\mathrm{L}}, s_i^{\mathrm{R}}]$, where $s_i^{\mathrm{L}}, s_i^{\mathrm{R}} \in S$ are the closest free servers on the line to the left and right of r_i, respectively, at the time of its arrival. For this to be well-defined, we possibly need to place an additional server at ∞ or $-\infty$. Denote by $P_i^{\mathrm{L}}, P_i^{\mathrm{R}}$ the alternating paths connecting r to s_i^{L} and s_i^{R}, respectively, that have shortest t-net cost. Recall that for the t-net-cost algorithm there are no free servers inside the search interval. Thus, we have $\bar{\mathrm{I}}_i \subseteq \mathrm{I}_i$ and the following observation holds.

Observation 1.21. *We have* $P_i \in \{P_i^{\mathrm{L}}, P_i^{\mathrm{R}}\}$.

The following lemma establishes some properties of the offline t-net-cost algorithm on alternating instances.

Lemma 1.22. *(i) Paths* $P_i^{\mathrm{L}}, P_i^{\mathrm{R}}$ *and matching* M_i^* *only use minimal edges.*

(ii) If $P_i = P_i^X$ *for* $X \in \{\mathrm{L}, \mathrm{R}\}$, *on* I_i, *locally, we have that* $M_i^* = M^{\mathrm{X}}$. *Specifically, this implies* $\phi_t^{M^*_{i-1}}(P_i) = \phi_t^{M^Y}(P_i)$ *for* $X \neq Y \in \{\mathrm{L}, \mathrm{R}\}$.

(iii) If P_j *is a child of* P_i, *then* $P_i = P_i^{\mathrm{L}}$ *if and only if* $P_j = P_j^{\mathrm{R}}$. *In particular, we have* $\mathrm{I}_j \subseteq P_i$ *and* $\mathrm{I}_j \cap M_j^* \cap M_i^* = \emptyset$.

Proof. Due to symmetry, we assume without loss of generality that $P_i = P_i^{\mathrm{R}}$.

(i): We use induction on i. For the base case, $i = 1$, Statement (i) is easily seen to be true. Hence, consider $i \geq 2$. Suppose, for sake of contradiction, that P_i

contains a non-minimal edge. Given a choice, consider the non-minimal edge e which is closest to s_i on P_i. Moving from r_i to s_i along P_i, edges leading from a server to a request are in M_{i-1}^* and by induction hypothesis minimal. Thus, $e = (r, s) \in M_i^*$ leads from r to s.

Consider the server s' that is directly to the right of r. Since e is not minimal, we have $r < s' < r' < s$, with r' being the request right of s'. We first consider the case that s' is free. Observation 1.21 implies $s' = s_i$. We show that altering P_i to go from r directly to s' yields a path of strictly lower t-net cost. To this end, first note that the remaining edges on P_i after s are minimal since e is closest to s_i. Thus, the t-net cost of the subpath of P_i starting at r is at least $t \cdot c(r, s) - c(s', s) = t \cdot c(r, s') + (t-1) \cdot c(s', s) > t \cdot c(r, s')$, with the last term being the t-net cost of connecting r directly to s'.

In the case that s' is matched, on the other hand, it must be matched to r' due to the induction hypothesis. Therefore, replacing e by $(r, s'), (s', r'), (r', s)$ reduces the t-net cost by $(t+1) \cdot c(s', r')$, contradicting the minimality of P_i.

(ii): By Statement (i), P_i^{L} and P_i^{R} only consist of minimal edges. Moving along P_i from left to right, edges from server to request are in M_{i-1}^* and edges from request to server in M_i^*. This is simply due to the fact that P_i is a path that augments M_{i-1}^* to form M_i^*, and it starts at a request and ends at a server. Therefore, $M_{i-1}^* \cap P_i^{\mathrm{R}} \subseteq M^{\mathrm{L}}$. With a symmetrical argument, we obtain $P_i^{\mathrm{L}} \cap M_{i-1}^* \subseteq M^{\mathrm{R}}$. After augmenting M_{i-1}^* along P_i, the edges point in the opposite direction and the claim follows.

(iii): By (ii), we know that $M_j^* \cap \mathrm{I}_j = M^{\mathrm{X}}$, say $X = R$, so edges are of the form (r, s) with $r < s$. Due to Statement (i), only augmenting paths leading from right to left can augment along such edges. If this happens, all of I_j is traversed as there is no free server in its interior. As parent of P_j, that path P_i is the first path to properly contain P_j and thus $P_i = P_i^{\mathrm{L}}$, and in particular, $\mathrm{I}_j \subseteq P_i$. The equation $\mathrm{I}_j \cap M_j^* \cap M_i^* = \emptyset$ then follows directly from statement (ii). □

Note that due to Lemma 1.22 (i), the sum of edge costs of an augmenting path is equal to the length of the corresponding line segment. In particular, we have $c(P_i^{\mathrm{L}}) + c(P_i^{\mathrm{R}}) = |\mathrm{I}_i|$. For simplicity, we abuse notation and use I_i to denote both, the interval on the line, and the set of edges $P_i^{\mathrm{L}} \cup P_i^{\mathrm{R}}$.

Lemma 1.22 can further be used to show that when considering the sum of lengths of augmenting paths of a certain depth in the path forest, then this number increases exponentially as the depth decreases, i.e., towards a root. Fix an augmenting path P_h and consider the induced subtree of augmenting paths with root P_h. Denote by $\mathcal{H}_k$ the set of indices of paths at depth k with P_h being at depth 0.

Lemma 1.23. *Consider a path P_h and its grandchildren, i.e., P_j with $j \in \mathcal{H}_2$. Then*

$$c(P_h) \geq \left(2 - \tfrac{1}{t}\right) \cdot \sum_{j \in \mathcal{H}_2} c(P_j).$$

Proof. Denote by P_i, for $i \in \mathcal{I} = \mathcal{H}_1$, the children of P_h in the path forest and by $\mathcal{J}_i \subseteq \mathcal{J} = \mathcal{H}_2$ the sets of indices of their respective children. Without loss of generality, assume that $P_h = P_h^{\mathrm{R}}$. Lemma 1.22 (iii), implies $P_i = P_i^{\mathrm{L}}$, and $P_j = P_j^{\mathrm{R}}$, for $i \in \mathcal{I}, j \in \mathcal{J}$; see Figure 1.6. With, again, Lemma 1.22 (iii), and using the fact that $\phi_t^{M_{i-1}^*}(P_i^{\mathrm{R}}) \geq \phi_t^{M_{i-1}^*}(P_i^{\mathrm{L}})$, by definition of the t-net-cost algorithm, we get

$$\begin{aligned} t \cdot c(P_h) &\geq t \cdot \sum_{i \in \mathcal{I}} \left(c(P_i^{\mathrm{L}}) + c(P_i^{\mathrm{R}})\right) \\ &\geq t \cdot \sum_{i \in \mathcal{I}} c(P_i^{\mathrm{L}}) + \sum_{i \in \mathcal{I}} \phi_t^{M_{i-1}^*}(P_i^{\mathrm{R}}) \\ &\geq t \cdot \sum_{j \in \mathcal{J}} c(P_j^{\mathrm{R}}) + t \cdot \sum_{i \in \mathcal{I}} c(P_i^{\mathrm{L}} \setminus (\cup_{j \in \mathcal{J}_i} P_j^{\mathrm{R}})) + \sum_{i \in \mathcal{I}} \phi_t^{M_{i-1}^*}(P_i^{\mathrm{L}}). \end{aligned} \tag{1.12}$$

Using Lemma 1.22 (ii), we get

$$\begin{aligned} \phi_t^{M_{i-1}^*}(P_i^{\mathrm{L}}) &\overset{\text{(ii)}}{=} \phi_t^{M^{\mathrm{R}}}(P_i^{\mathrm{L}}) \\ &= \sum_{j \in \mathcal{J}_i} \left(\phi_t^{M^{\mathrm{R}}}(P_j^{\mathrm{L}}) + \phi_t^{M^{\mathrm{R}}}(P_j^{\mathrm{R}})\right) + \phi_t^{M^{\mathrm{R}}}(P_i \setminus (\cup_{j \in \mathcal{J}_i} \mathrm{I}_j)) \\ &\overset{\text{(ii)}}{\geq} \sum_{j \in \mathcal{J}_i} \left(\phi_t^{M_{j-1}^*}(P_j^{\mathrm{L}}) + \phi_t^{M^{\mathrm{R}}}(P_j^{\mathrm{R}})\right) - t \cdot c(P_i \setminus (\cup_{j \in \mathcal{J}_i} \mathrm{I}_j)). \end{aligned}$$

Since $\phi_t^{M_{j-1}^*}(P_j^{\mathrm{L}}) \geq \phi_t^{M_{j-1}^*}(P_j^{\mathrm{R}}) = \phi_t^{M^{\mathrm{L}}}(P_j^{\mathrm{R}})$, the above together with Inequality (1.12) implies

$$t \cdot c(P_h) \geq \sum_{j \in \mathcal{J}} \left(t \cdot c(P_j^{\mathrm{R}}) + \phi_t^{M^{\mathrm{L}}}(P_j^{\mathrm{R}}) + \phi_t^{M^{\mathrm{R}}}(P_j^{\mathrm{R}}) \right) \geq (2t-1) \cdot \sum_{j \in \mathcal{J}} c(P_j^{\mathrm{R}}),$$

where last inequality follows from the observation that, due to Equation (1.1), it holds that $\phi_t^{M^{\mathrm{L}}}(P) + \phi_t^{M^{\mathrm{R}}}(P) = (t-1) \cdot c(P)$. □

We are now equipped to prove Theorem 1.3.

Theorem 1.3. *For alternating instances of online minimum-cost matching on the line, there is a $(1+\varepsilon)$-competitive algorithm that reassigns each request $O(\frac{1}{\varepsilon^2})$ times.*

Proof. Fix an augmenting path P_h and consider its descendants P_j which are at depth $2k+2$ in the path tree rooted at P_h, i.e., $j \in \mathcal{H}_{2k+2}$. The intervals I_j are contained in paths $P_{j'}$, $j' \in \mathcal{H}_{2k+1}$, by Lemma 1.22 (iii). Lemma 1.23 implies

$$\sum_{j \in \mathcal{H}_{2k+2}} |I_j| \leq \sum_{j' \in \mathcal{H}_{2k+1}} c(P_{j'}) \leq \left(2 - \tfrac{1}{t}\right)^{-k} \cdot \sum_{i \in \mathcal{H}_1} c(P_i). \tag{1.13}$$

Raghvendra [Rag16] shows that the t-net cost of augmenting paths is always non-negative. In particular, $\phi_t^{M_{h-1}^*}(P_h) = t \cdot c(P_h \cap M_h^*) - c(P_h \cap M_{h-1}^*) \geq 0$. Therefore, we obtain

$$\sum_{i \in \mathcal{H}_1} c(P_i) \leq c(P_h) = c(P_h \cap M_h^*) + c(P_h \cap M_{h-1}^*) \leq (t+1) \cdot c(P_h \cap M_h^*). \tag{1.14}$$

Similarly, we have

$$\sum_{i \in \mathcal{H}_1} c(P_i) \leq (t+1) \cdot \sum_{i \in \mathcal{H}_1} c(P_i \cap M_i^*) \leq (t+1) \cdot c(P_h \cap M_i^*). \tag{1.15}$$

On an interval I_i with $i \in \mathcal{H}_1$, locally, $M_h^* = M^{\mathrm{L}}$ if and only if $M_i^* = M^{\mathrm{R}}$ by Lemma 1.22 (ii). For $X = \mathrm{L}$ and $X = \mathrm{R}$, Equation (1.13) together with Equation (1.14) and (1.15), respectively, implies

$$\sum_{j \in \mathcal{H}_{2k+2}} |\mathrm{I}_j| \leq \left(2 - \tfrac{1}{t}\right)^{-k} \cdot (t+1) \cdot c\left(P_h \cap M^{\mathrm{X}}\right).$$

Rearranging, we obtain

$$\left[\tfrac{1}{t+1}\left(2 - \tfrac{1}{t}\right)^{k} - 1\right] \cdot \sum_{j \in \mathcal{H}_{2k+2}} |\mathrm{I}_j| \leq c\left(P_h \setminus (\cup_{j \in \mathcal{H}_{2k+2}} \mathrm{I}_j) \cap M^{\mathrm{X}}\right). \tag{1.16}$$

Denote by $\alpha(k,t)$ the term in square brackets. When a (minimal) edge (r,s) is frozen, the remaining instance on $S \setminus \{s\} \times R \setminus \{r\}$ is again alternating and at most one request will take a detour due to (r,s) being frozen. The additionally incurred cost when compared to M^* is bounded by $2 \cdot |\mathrm{I}_r|$ and can, via Inequality (1.16), be charged to non-frozen parts of M^{OPT} (recall that $M^{\mathrm{OPT}} \in \{M^{\mathrm{L}}, M^{\mathrm{R}}\}$). The set difference $P_h \setminus (\cup_{j \in \mathcal{H}_{2k+2}} \mathrm{I}_j)$ in Inequality (1.16) ensures that the portion $P_h \cap M^{\mathrm{OPT}}$ of M^{OPT} is charged at most $2k+2$ times before P_h itself is frozen and the cost is charged to another portion of M^{OPT}. This leads to a competitive ratio of $(t + \frac{2k+2}{\alpha(k,t)})$.

We set $t = 1 + \varepsilon$ and show that this results in a competitive ratio of $1 + 2\varepsilon$ for an appropriate choice of k. This is equivalent to showing that

$$2k + 2 \leq \varepsilon \cdot \alpha(k,t) = \frac{\varepsilon}{2+\varepsilon} \cdot \left(1 + \frac{\varepsilon}{1+\varepsilon}\right)^{k} - \varepsilon.$$

Assuming $\varepsilon < 2$ and using $\left(1 + \frac{\varepsilon}{1+\varepsilon}\right)^{\frac{4}{\varepsilon}} \geq e$ for $\varepsilon < 2$, this can be further simplified to

$$k \cdot \frac{10}{\varepsilon} \leq (e^{\frac{\varepsilon}{4}})^{k}.$$

After setting $k = 30 \cdot \varepsilon^{-2}$, this statement is true for all $\varepsilon < 2$. This concludes the proof of the theorem. □

As a byproduct, we show, for this special class of instances, a result in the online setting without recourse. It relates the competitive ratio to the cost metric, i.e., the maximum difference in edge cost for connecting a request to

a server and refines the best known competitive ratio of $O(\log n)$ by Raghvendra [Rag18]. The proof of the following theorem additionally contains a proof for an upper bound of $O(\log n)$ for alternating instances that is much simpler than the general proof by Raghvendra [Rag18].

Theorem 1.24. *For alternating instances, the online t-net-cost algorithm is $O(\min\{\log n, \log \Delta\})$-competitive, where $\Delta = \max_{r,r' \in R, s,s' \in S} \frac{c(r,s)}{c(r',s')}$.*

Proof. Consider an edge e_i of M' and assume $P_i = P_i^{\mathrm{R}}$. By Lemma 1.22 (ii) and the definition of the t-net-cost algorithm,

$$\begin{aligned} t \cdot c(P_i^{\mathrm{L}} \cap M^{\mathrm{L}}) - c(P_i^{\mathrm{L}} \cap M^{\mathrm{R}}) &= \phi_t^{M_{i-1}^*}(P_i^{\mathrm{L}}) \\ &\geq \phi_t^{M_{i-1}^*}(P_i^{\mathrm{R}}) \\ &= t \cdot c(P_i^{\mathrm{R}} \cap M^{\mathrm{R}}) - c(P_i^{\mathrm{R}} \cap M^{\mathrm{L}}), \end{aligned}$$

which implies

$$c(P_i^{\mathrm{R}} \cap M^{\mathrm{R}}) \leq \tfrac{1}{t} \cdot c(P_i^{\mathrm{R}} \cap M^{\mathrm{L}}) + c(P_i^{\mathrm{L}} \cap M^{\mathrm{L}}).$$

Therefore,

$$\begin{aligned} c(P_i) &= c(P_i^{\mathrm{R}} \cap M^{\mathrm{L}}) + c(P_i^{\mathrm{R}} \cap M^{\mathrm{R}}) \\ &\leq \left(1 + \tfrac{1}{t}\right) \cdot c(P_i^{\mathrm{R}} \cap M^{\mathrm{L}}) + c(P_i^{\mathrm{L}} \cap M^{\mathrm{L}}). \end{aligned} \tag{1.17}$$

As in Equation (1.14), we use $\phi_t^{M_{i-1}^*}(P_i) = t \cdot c(P_i \cap M_i^*) - c(P_i \cap M_{i-1}^*) \geq 0$ to obtain

$$\begin{aligned} c(P_i) &= c(P_i \cap M_i^*) + c(P_i \cap M_{i-1}^*) \\ &\leq (t+1) \cdot c(P_i \cap M_i^*) \\ &= (t+1) \cdot c(P_i \cap M^{\mathrm{R}}). \end{aligned} \tag{1.18}$$

Together, Inequalities (1.17) and (1.18) imply

$$c(e_i) = c(P_i) \leq c(\mathrm{I}_i \cap M^{\mathrm{OPT}}) \cdot \max\{t+1, 1+\tfrac{1}{t}\}.$$

Since intervals with have the same depth in the path forest are disjoint, we can bound the lengths of paths with identical depth by $\max\{t+1, 1+\frac{1}{t}\} \cdot \mathrm{OPT}$. As

there are at most $2 \cdot \log_{(2-1/t)} \Delta$ different depths in total due to Lemma 1.23, this bounds the competitive ratio by $O(\log \Delta)$.

Further, by using the same approach as above, we can directly obtain the upper bound of $O(\log n)$ from Raghvendra [Rag18]. Specifically, we can bound the cost of the nodes of depth $O(\log n)$ or lower in the path forest by $O(\log n) \cdot \max\{t+1, 1+\frac{1}{t}\} \cdot \mathrm{OPT}$ as above. And for nodes P_j of larger depth, we denote by P_i the root of the corresponding path tree and apply Lemma 1.23 repeatedly, $O(\log n)$ times, to bound their cost by

$$c(P_j) \leq O(\tfrac{1}{n}) \cdot c(P_i) \leq \tfrac{1}{n} \cdot O(\mathrm{OPT}).$$

Noting that there are less than n such edges completes the proof. □

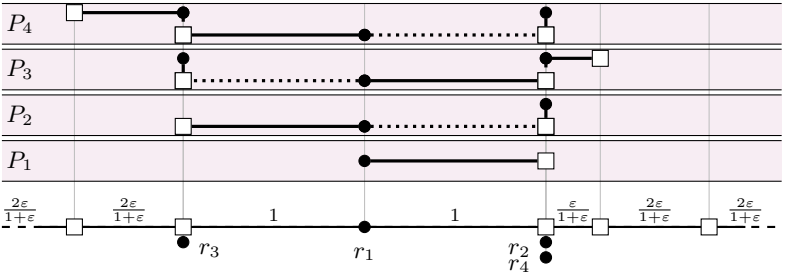


Fig. 1.7: A non-alternating instance, where the strategy from Section 1.3 fails. Edge costs are indicated above the corresponding portion of the line. Paths P_i are the minimum t-net-cost paths chosen by the offline matching M^* for augmentation ($t = 1 + \varepsilon$). In each iteration, request r_1 is reassigned. However, the cost of the optimum solution does not increase exponentially. More precisely, $\mathrm{OPT}_i = 1 + \sum_{j=1}^{i-1} \frac{j \cdot \varepsilon}{1+\varepsilon}$. It is therefore not possible to freeze request r_1 after a constant number of reassignments. Note, however, that the amortized recourse budget is constant, even without freezing.

1.4 Conclusion

In this chapter, we gave first non-trivial results for the online minimum-cost bipartite matching problem with recourse. The results were obtained simultaneously with and independently of Gupta et al. [GKS20] who consider also more general metrics than the line. We confirmed that an average

recourse of $O(\log n)$ per request is sufficient to obtain an $O(1)$-competitive matching on the line. It remains open if such a result can be obtained in a non-amortized setting, where the recourse is available only per iteration. Our algorithm is clearly designed for the amortized setting as it buffers online matching decisions and repairs them in batches.

Further, it remains open whether constant recourse per request is sufficient for maintaining an $O(1)$-competitive matching on the line, as for the case of alternating requests. This may be very well possible as there is, currently, no lower bound that rules this out. Note, that the strategy proposed in Section 1.3 fails in the general case, see Figure 1.7 for an example. However, we hope that the ideas provided in this chapter still prove helpful for solving this open question.

Fully Dynamic Algorithms for Knapsack Problems with Polylogarithmic Update Time

2

Knapsack problems are among the most fundamental optimization problems. In their most basic form, we are given a knapsack with capacity $S \in \mathbb{N}$ and a set of n items, where each item $j \in [n] := \{1, 2, \ldots, n\}$ has a size $s_j \in \mathbb{N}$ and a value $v_j \in \mathbb{N}$. The *knapsack* problem asks for a subset of items, $P \subseteq [n]$, with maximum total value $v(P) := \sum_{j \in P} v_j$ and such that the total size $s(P) := \sum_{j \in P} s_j$ 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 $P_1, P_2, \ldots, P_m \subseteq [n]$ that satisfy the capacity constraints $s(P_i) \leq S_i$ and are such that the total value of all subsets $\sum_{i \in [m]} v(P_i)$ is maximized.

Regarding the computational complexity of knapsack variants, it is known that multiple knapsack is strongly $\mathcal{NP}$-hard, even for identical knapsack capacities, as it is a special case of bin packing. The knapsack problem, on the other hand, is only weakly $\mathcal{NP}$-hard and admits pseudo-polynomial time algorithms, the first one of which was already published in the 1950s [Bel57].

As a consequence of these hardness results, each of the knapsack variants has been studied extensively through the lens of approximation algorithms. Of particular interest are *approximation schemes*, families of polynomial-time algorithms that contain an algorithm for each $\varepsilon > 0$ that computes a $(1 - \varepsilon)$-approximate solution, i.e., a feasible solution with total value within a factor of $(1 - \varepsilon)$ of that of an optimal solution OPT, in time polynomial in the input size. Based on the dependency on ε of the running time, we distinguish *Polynomial Time Approximation Schemes (PTAS)* with arbitrary dependency on ε, i.e., a running time of $n^{f(\varepsilon)}$, *Efficient PTAS (EPTAS)* where arbitrary functions $f(\varepsilon)$ may only appear as a multiplicative factor, i.e., $f(\varepsilon)n^{O(1)}$, and *Fully Polynomial Time Approximation Schemes (FPTAS)* with polynomial dependency on $\frac{1}{\varepsilon}$, i.e., $\frac{1}{\varepsilon}^{O(1)} n^{O(1)}$.

The first approximation scheme for knapsack was an FPTAS by Ibarra and Kim [IK75] and initiated a long sequence of follow-up work, which is still active [Cha18; Jin19]. multiple knapsack is substantially harder and does not admit an FPTAS, unless $\mathcal{P} = \mathcal{NP}$, even with two identical knapsacks [CK05]. However, approximation schemes with running times of the form $n^{f(\varepsilon)}$ (PTASs) are known [Kel99; CK05] as well as improvements to only $f(\varepsilon)n^{O(1)}$ (EPTASs) [Jan09; Jan12]. All these algorithms are *static* in the sense that the full instance is given to an algorithm and is then solved.

Given the ubiquitous dynamics of real-world problems, many of which can be modeled as knapsack problems, it is natural to ask for *dynamic algorithms* that adapt to small changes in the packing instance while spending only little computation time. More precisely, during the execution of the algorithm, items and knapsacks arrive and depart and the algorithm needs to maintain an approximate knapsack solution with an *update time* polylogarithmic in the number of items in each step. A dynamic algorithm is then equivalent to a data structure that implements these updates efficiently and supports relevant query operations such as asking for the packing of a specific or for the solution value.

A practical application is the dynamic estimation of the profit for scheduling jobs in computing clusters in which virtual machines can be moved among physical machines [BB10]. This allows the service provider to adapt the provided capacity, i.e., the currently running servers, to the current demand, see, e.g., [BKB07; LTC14; DKL14]. An efficient framework for multiple knapsack can be viewed as a first-stage decision tool: In real-time, it determines whether the customer in question should be allowed into the system based on the cost of possibly powering and using additional servers. As the service provider has to decide immediately which request she wants to accept, she needs to obtain the information *fast*, i.e., sublinear in the number of requests already in the system.

Generally, dynamic algorithms constitute a vibrant research field in the context of graph problems. We refer to surveys [DEG+10; Hen18; BP11] for an overview on dynamic graph algorithms. Interestingly, dynamic algorithms with *polylogarithmic* update time are known only for a small number of graph problems, among them connectivity problems [HK99; HLT01],

the minimum spanning tree problem [HLT01], and the vertex cover problem [BHN17; BK19]. Recently, this was complemented by multiple conditional lower bounds that are typically *linear* in the number of nodes or edges; see, e.g., [AW14]. Over the last few years, the generalization of dynamic vertex cover to dynamic set cover gained interest leading to near-optimal approximation algorithms with polylogarithmic update times [BHN19; BHI15; GKK+17; AAG+19]. Also recently, algorithms have been developed for maintaining maximal independent sets, e.g., [BDH+19; CZ19; Mon19], and approximate maximum independent sets in special graph classes [HNW20; CMR20; BCI+20].

For packing problems, there are hardly any dynamic algorithms with small update time known. A notable exception is a result for bin packing that maintains a $\frac{5}{4}$-approximate solution with $O(\log n)$ update time [IL98]. This lack of efficient dynamic algorithms is in stark contrast to the aforementioned intensive research on computationally efficient algorithms for packing problems. Our work bridges this gap initiating the design of data structures and algorithms that efficiently maintain near-optimal solutions.

Our Contribution In this chapter, we present dynamic algorithms for maintaining approximate solutions for three problems of increasing complexity: knapsack, multiple knapsack with only few knapsacks, and general multiple knapsack without any restrictions. Our algorithms are *fully dynamic* which means that in an update operation they can handle the arrival or departure of an item or 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, which would imply 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 for the packing of an item – i.e., was the queried item packed in the solution, and if so, into which knapsack? – or for the value of the computed solution can be answered in polylogarithmic time.

We give *worst-case* guarantees for update and item query times that are polylogarithmic 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 superpolynomial dependency on $\frac{1}{\varepsilon}$ with suitable lower bounds. We remark that it is not possible to maintain a solution with a non-trivial approximation guarantee explicitly with only polylogarithmic 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. Further, we remark that in the static setting, our result yields an algorithm with running time near-linear in n. Note that, in the context of dynamic algorithms, n and m refer to the number of current items and knapsacks, respectively.

Denote by $\overline{v}$ the currently largest item value, by $\overline{\overline{v}}$ a known upper bound on $\overline{v}$, and by $S_{\max}$ the currently largest capacity of any knapsack.

(i) For multiple knapsack, we give a fully dynamic algorithm that maintains $(1-\varepsilon)$-approximate solutions with an update time of at most $(\frac{\log n}{\varepsilon})^{O(\frac{1}{\varepsilon})} \cdot (\log \overline{v} \cdot \log S_{\max})^{O(1)}$. Single items can be queried in time $O(\log(\frac{\log n}{\varepsilon}))$, the entire solution P in time $O(\log(\frac{\log n}{\varepsilon})|P|)$, and the solution value in time $O(1)$ (Section 2.5).

(ii) The dependency on $\frac{1}{\varepsilon}$ in the update time for multiple knapsack does indeed need to be superpolynomial, even for two identical knapsacks. We show that there is no $(1-\varepsilon)$-approximate dynamic algorithm with update time $(\frac{\log n}{\varepsilon})^{O(1)}$, unless $\mathcal{P} = \mathcal{NP}$ (Section 2.1).

(iii) For multiple knapsack with the number of knapsacks only polylogarithmic in n, we give a dynamic algorithm with improved performance that has an update time of $2^{O(\frac{1}{\varepsilon}\log^4(\frac{1}{\varepsilon}))} \cdot (\frac{\log(n\overline{v})}{\varepsilon})^{O(1)} + O(\frac{1}{\varepsilon}\log n \log \overline{\overline{v}})$. Item queries are answered in time $O(\log \frac{\log n}{\varepsilon})$, solution-value queries in time $O(1)$, and queries of one knapsack or the entire solution in time linear in the number of packed items therein (Section 2.3).

(iv) Lastly, we present a particularly efficient algorithm for the knapsack problem. It maintains $(1-\varepsilon)$-approximate implicit solutions and has an update time of $(\frac{\log(n\overline{v})}{\varepsilon})^{O(1)} + O(\frac{1}{\varepsilon}\log n \log \overline{\overline{v}})$. Surprisingly, the query times of the algorithm are essentially equivalent to the time it takes to access an explicit solution from memory (Section 2.2).

In each update step, we only compute implicit solutions. Moreover, we provide query operations to obtain the solution value, the packing of a

queried item and the complete solution. These queries run efficiently, i.e., in time polynomial in $\log n$ and $\log \overline{\overline{v}}$ with additional dependencies on $\frac{1}{\varepsilon}$ and the output size. Furthermore, the queries are consistent between two update steps, by which we mean that there exists a single explicit solution which is consistently revealed via the provided queries. Specifically, while previous queries can influence whether and into which knapsack an item is packed, once the item is queried these properties are fixed and do not change during subsequent queries before the next update step.

Our Techniques Maybe surprisingly, we recompute a $(1-\varepsilon)$-approximate solution from scratch in polylogarithmic time after each update. More precisely, we compute a $(1-\varepsilon)$-estimate of the value of OPT and additionally store all information that is needed in order to answer any query in polylogarithmic time. Interestingly, this shows that, for such computations, we do not need exact knowledge about the whole input, but only a small amount of information of polylogarithmic size. We show that this information can be extracted efficiently from suitable data structures in which we store the input items and knapsacks. We also show that we can maintain these data structures in polylogarithmic time per update.

On a high level, when solving multiple knapsack, we reduce the overall problem to two subproblems that are solved independently. In the first one, we deal with only few knapsacks, i.e., $m = (\frac{\log n}{\varepsilon})^{O(1)}$ many, which are the largest knapsacks in the original input. Here, we observe that, if we select the $\frac{m}{\varepsilon}$ most valuable items in the optimal solution correctly, we can afford to fill the remaining space in the knapsacks greedily, i.e., highest density (value divided by size) first, and charge the resulting loss to the valuable items. We cannot guess these most valuable items explicitly, but we show that we can select a small set of candidates for these items and guess a few placeholder items for the remaining ones. This yields an instance with only $(\frac{\log n}{\varepsilon})^{O(1)}$ items on which we run a known EPTAS for multiple knapsack due to Jansen [Jan12] so that the total update time is $2^{O(\frac{1}{\varepsilon}\log^4(\frac{1}{\varepsilon}))} \cdot (\frac{\log(n\overline{v})}{\varepsilon})^{O(1)}$. For the special case of a single knapsack, we show that we can invoke an FPTAS instead of an EPTAS to further improve the running time.

In the second subproblem, we handle a potentially large set of knapsacks, and we are allowed to use an additional set of $(\frac{\log n}{\varepsilon})^{\Theta(1)}$ knapsacks that

the optimal solution does not use (resource augmentation). We introduce a technique that we call *oblivious linear grouping*. Linear grouping is a standard technique used in order to round a set of one-dimensional items that need to be packed into a given set of containers (e.g., in bin packing), such that they have at most $\frac{1}{\varepsilon}$ different sizes after the rounding (at the expense of leaving an ε-fraction of the items out). However, in our setting we do not know a priori which input items need to be packed, and therefore we cannot apply this technique directly. Instead, we show that we can round the input items to $(\frac{\log n}{\varepsilon})^{O(1)}$ different sizes such that we lose at most a factor of $(1-\varepsilon)$ *independently* of what the optimal solution looks like. In fact, our rounding method is even oblivious to the input knapsacks. Therefore, we believe that it might be useful also for other dynamic packing problems or for speeding up static algorithms. After rounding the items to $(\frac{\log n}{\varepsilon})^{O(1)}$ different sizes, we set up a configuration LP that has a configuration for each possible set of relatively large items that together fit into a knapsack. Thanks to our rounding, there are only polylogarithmically many configurations and we can solve this LP in time $(\frac{\log n}{\varepsilon})^{O(\frac{1}{\varepsilon})}$. We use the additional knapsacks in order to compensate for errors that may occur when rounding the LP solution, i.e., due to rounding up fractional variables and adding small items greedily into the remaining space of the knapsacks. Special care is necessary since the sizes of the knapsacks can differ and hence some item might be relatively large in some knapsack but relatively small in another knapsack.

Further Related Work Since the first approximation scheme for knapsack [IK75], running times have been improved steadily [GL79; Law79; GL80; KP04; Rhe15; Cha18; Jin19] with $O(n \log \frac{1}{\varepsilon} + (\frac{1}{\varepsilon})^{9/4})$ by Jin [Jin19] being the currently fastest. Recent work on conditional lower bounds [CMW+19; KPS17] implies that knapsack does not admit a FPTAS with running time $O((n+\frac{1}{\varepsilon})^{2-\delta})$, for any $\delta > 0$, unless $(\min, +)$-convolution has a subquadratic-time algorithm [MWW19; Cha18].

A PTAS for multiple knapsack was first presented by Chekuri and Khanna [CK05], and EPTASs due to Jansen [Jan09; Jan12] are also known. The fastest of these has a running time of $2^{O(\frac{1}{\varepsilon}\log^4(\frac{1}{\varepsilon}))} + n^{O(1)}$ [Jan12]. The mentioned algorithms are all static and assume full knowledge about the instance for which a solution has to be found. In particular, their solutions

might change completely when a single item is added to the input which makes a full recomputation necessary. The algorithm in [CK05] involves a guessing step with $n^{f(\frac{1}{\varepsilon})}$ options, too many for a polylogarithmic update time. The EPTASs in [Jan09; Jan12] use a configuration linear program of size $\Omega(n)$, which is also prohibitively large for the dynamic setting.

The dynamic arrival and removal of items exhibits some similarity to knapsack models with incomplete information. For example, in the *online* knapsack problem [MV95] items arrive online one by one. When an item arrives, an algorithm must irrevocably accept or reject it before the next item arrives. Various problem variants have been studied, e.g., with resource augmentation [IZ10], the removable online knapsack problem [IT02; HM10; HKM+14; HKM13; CJS16], and with advice [BKK+14]. Other models with uncertainty in the item set or the knapsack capacity include the *stochastic* knapsack problem [DGV08; BGK11; Ma18] and *robust* knapsack problems [Yu96; MM13; DKM+17; BKK11]. Related to our setting, there are also online models with a softened irrevocability requirement, e.g., online optimization with *recourse* [MSV+16; IW91; GGK16; FFG+18] or *migration* [SSS09; SV16; JK19] allows to adapt previously taken decisions in a limited way. We are not aware of work on knapsack problems in these settings and, again, in all these settings, the goal is to bound the amount of change needed when updating a solution regardless of the computational effort.

Structure of the Chapter As a necessary basis for all our algorithms, Section 2.1 formally defines the operations the algorithms support when viewed as a data structure, describes commonly used auxiliary data structures, and details a rounding procedure for item values that will be used throughout the chapter. Here, we also discuss the need for the exponential dependency on $\frac{1}{\varepsilon}$ in the update time for multiple knapsack. Then, in Section 2.2, we describe a very efficient algorithm for one knapsack and extend it in Section 2.3 to an algorithm for a polylogarithmic number of knapsacks. In Section 2.4, we present the oblivious linear grouping technique and an algorithm for multiple knapsack that relies on resource augmentation in the form of a polylogarithmic number of additional knapsacks. Finally, we present in Section 2.5 an algorithm for multiple knapsack without any restrictions that uses the algorithms from Sections 2.3 and 2.4 as subroutines.

2.1 Preliminaries and Data Structures

From the perspective of a data structure that implicitly maintains near-optimal solutions for multiple knapsack, our algorithms support several update and query operations which are listed below. They allow for the output of (parts of) the current solution and its value, or for the following two specific changes to the input of multiple knapsack, which cause the computation of a new solution.

- **Insert (Remove) Item:** Inserts (removes) item into (from) the input
- **Insert (Remove) Knapsack:** Inserts (removes) knapsack into (from) the input

The current solution and its value 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 this is the case, additionally returns the knapsack containing it
- **Query Solution Value:** Returns the value of the current solution
- **Query Entire Solution:** Returns indices of all items in the current solution, together with the information in which knapsack each such item is packed

Auxiliary Data Structures The targeted running times do not allow for completely reading the instance in every round but rather ask for carefully maintained auxiliary data structures that allow us to quickly compute and store implicit solutions. In these data structures, we store (subsets of) input items and input knapsacks, sorted according to some specific element property, e.g., size or capacity. Our requirements on the data structures are to enable the following operations in logarithmic running time: (i) access to elements, (ii) computation of the largest prefix of elements such that the prefix sum according to some property, e.g., the combined size of items in the prefix, is below a given threshold, and (iii) computation of a prefix sum over with respect to some element property, e.g., the total value of a prefix. Note that the prefixes are defined with respect to the fixed ordering of the elements, while the element properties for the threshold or computing the sum may be

arbitrary and, in particular, distinct from one another. When computing a largest prefix, we output the index of its last element.

The above requirements can be fulfilled by employing as auxiliary data structures a variation of balanced search trees. Specifically, we build on the work of Bayer and McCreight, who developed such a data structure in 1972, the so-called B-trees, which was later refined by Bayer to *symmetric binary B-trees*. In contrast to this early work, we additionally store for each node v several pieces of information such as the total size, the total value, the total number of elements, or the total capacity of the subtree rooted in v. 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. For our dynamic 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 his 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 logarithmic in the number of stored elements. Storing the additional values such as total size of a subtree in its root then also allows us to execute the aforementioned operations in logarithmic time. Additionally, we assume that we can iterate over the elements sequentially in the given ordering, which can also easily be implemented.

Lemma 2.1. *There is a data structure that maintains a linear order of n' elements with respect to some element property X and additionally allows for:*

(i) Insertion, deletion, or search by value with respect to property X or index of an element in time $O(\log n')$.
(ii) Computation of prefixes and prefix sums with respect to any element property in time $O(\log n')$.

Assumptions We provide a list of assumptions that for simplicity's sake will be made throughout this chapter.

- Elementary operations (e.g., additions) are computed in constant time.
- Without loss of generality, we have $\frac{1}{\varepsilon} \in \mathbb{N}$.

- We have $n \geq m$. Otherwise we may use only the n largest knapsacks.
- We assume access to perfect hash tables that provide $O(1)$ lookup times for explicitly saved items of our solution.
- When determining which of two items has the larger size (value), we break ties by choosing, if possible, the item with smaller value (size), and otherwise, that of higher item index. When comparing density, break ties by index.
- At the very beginning, we start with only the just mentioned dummy items and a single knapsack, and initialize all needed auxiliary data structures accordingly. If one wishes to start with a specific set of items and/or knapsacks, it is possible to simply insert them – one at a time – with our insertion routines, using polylogarithmic time per insertion.

Rounding of Values Another crucial ingredient for our algorithms is the partitioning of items into a small number of *value classes* V_ℓ, where for each ℓ the class V_ℓ consists of all input items j with $(1+\varepsilon)^\ell \leq v_j < (1+\varepsilon)^{\ell+1}$. Upon arrival of some item j, we calculate the index ℓ_j such that $j \in V_{\ell_j}$ and store the tuple (j, v_j, s_j, ℓ_j) representing j in the auxiliary data structures of the respective algorithm. In the remainder of this chapter, we assume, for each ℓ, that an item j in V_ℓ has value $(1+\varepsilon)^\ell$ and refer to this rounded value by v_j. We explicitly state, when we talk about non-rounded values. As stated in the following lemma, we hereby lose only a factor of $\frac{1}{1+\varepsilon}$ when considering the total profit of any solution. Since this technique is rather standard, we only state the lemma and omit a formal proof.

Lemma 2.2. *Recall that $\overline{v}$ is the currently largest item value and $\overline{\overline{v}}$ an upper bound on $\overline{v}$. After rounding item values to value classes as described above, the following holds.*

(i) There are at most $O(\frac{\log \overline{v}}{\varepsilon})$ non-empty value classes.

(ii) With respect to original values, two optimal solutions OPT and OPT' for the original and rounded instance, respectively, satisfy the inequality $v(\mathrm{OPT}') \geq (1-\varepsilon) \cdot v(\mathrm{OPT})$.

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. As Chekuri and Khanna [CK05] observed, multiple knapsack with $m = 2$ does not admit an FPTAS unless we have $\mathcal{P} = \mathcal{NP}$.

Theorem 2.3 (Proposition 2.1 in [CK05])**.** *If multiple knapsack with two identical knapsacks has an FPTAS, then partition can be solved in polynomial time. Hence there is no FPTAS for multiple knapsack even with $m = 2$, unless $\mathcal{P} = \mathcal{NP}$.*

For the dynamic setting, this implies that there is no dynamic algorithm with running time polynomial in $\log n$ and $\frac{1}{\varepsilon}$ unless $\mathcal{P} = \mathcal{NP}$. We are able to extend this result to the case where $m \leq \frac{1}{3\varepsilon}$. Specifically, we show that the existence of a $(1-\varepsilon)$-approximate dynamic algorithm for multiple knapsack with $m < \frac{1}{3\varepsilon}$ with update time polynomial in $\log n$ and $\frac{1}{\varepsilon}$ would imply that 3-partition can be decided in polynomial time. The statement focuses on dynamic algorithms, our main interest here, but the proof does not use the dynamic nature, only the final complexity. Also note that the result can be extended to a larger number of knapsacks by adding an appropriate number of sufficiently small knapsacks, i.e., polynomially many in n.

Theorem 2.4. *Unless $\mathcal{P} = \mathcal{NP}$, there is no algorithm for multiple knapsack that computes a $(1-\varepsilon)$-approximate solution in time polynomial in n and $\frac{1}{\varepsilon}$, for $m < \frac{1}{3\varepsilon}$.*

Proof. Consider the strongly $\mathcal{NP}$-hard problem *3-partition* [GJ79], where there are $3m$ items with sizes $a_j \in \mathbb{N}$ such that $\sum_{j=1}^{3m} a_j = mA$. We use the restricted-input variant where sizes belong to $(A/2,\ A/4)$ so that only subsets of cardinality 3 may sum to A. The task is to decide whether there exists a partition $cup_{i=1}^{m} J_i = [3m]$ such that $|J_i| = 3$ and $\sum_{j \in J_i} a_j = A$ for $1 \leq i \leq m$.

Consider the following instance for multiple knapsack: There are m knapsacks with capacity $S = A$ and $3m$ items. Each item corresponds one-to-one to an item in the 3-partition instance with $s_j = a_j$ and $v_j = 1$ for $1 \leq j \leq 3m$. Observe that the 3-partition instance is a YES-instance if and only if the optimal solution to the knapsack problem contains $3m$ items. In such a solution, each knapsack must contain exactly 3 items.

Assume multiple knapsack admits an algorithm with approximation factor at least $(1-\varepsilon)$ and running time polynomial in $\frac{1}{\varepsilon}$ and n where $m < \frac{1}{3\varepsilon}$. Such an algorithm is then able to obtain solutions with an objective function larger than $3m-1$ for multiple knapsack instances created from 3-partition instances. As this implies an (optimal) objective value of $3m$, such an algorithm decides 3-partition in polynomial time, which is not possible, unless $\mathcal{P} = \mathcal{NP}$. □

2.2 A Single Knapsack

We start the presentation of our results with the first dynamic algorithm for the most simple knapsack variant considered in this chapter, the knapsack problem. Beyond serving as a convenient warm-up, this problem is unique in that we obtain an algorithm with fully polynomial update time as well as constant item query times. The latter is comparable to having access to an explicit solution. Recall that $\overline{v}$ is the currently largest item value and $\overline{\overline{v}}$ an upper bound on $\overline{v}$.

Theorem 2.5. *For $\varepsilon > 0$, there is a fully dynamic algorithm for knapsack that maintains $(1-\varepsilon)$-approximate solutions with update time $O(\frac{1}{\varepsilon^8}\log^4(n\overline{v})) + O(\frac{1}{\varepsilon}\log n \log \overline{\overline{v}})$. Furthermore, single items and the solution value can be queried in time $O(1)$, the entire solution P in time $O(|P|)$.*

On a high level, the update procedure of our algorithm works as follows. Consider a partition of the items in some fixed optimal solution OPT into high-value and low-value items, with the high-value items being the $\frac{1}{\varepsilon}$ most valuable items of OPT, denoted by $\mathrm{OPT}_{\frac{1}{\varepsilon}}$, and the low-value items being the remaining ones. We compute a small set of candidate items $H_{\frac{1}{\varepsilon}}$ that contains all relevant high-value items that could possibly be in $\mathrm{OPT}_{\frac{1}{\varepsilon}}$. Additionally, we create a placeholder item as a stand-in for the low-value items in OPT and guess its size so that it is large enough to be able to fractionally accommodate low-value items with corresponding total value. An optimal fractional solution of low-value items can be obtained by iteratively packing the items with the highest *density*, i.e., $\frac{v_j}{s_j}$ for item j, until one item does not fit, and then pack this item fractionally. Since this is the only item that is packed fractionally,

we can discard it and charge the resulting loss in value to the $\frac{1}{\varepsilon}$ high-value items. This results in a knapsack instance with only $O\left(\frac{1}{\varepsilon^3}\right)$ items which we solve with an FPTAS to obtain the desired guarantees.

Notation and Data Structures Before describing the details of this procedure, we fix some notation and describe the necessary auxiliary data structures. Fix an optimal solution OPT and denote by $\mathrm{OPT}_{\frac{1}{\varepsilon}}$ a set of the $\min\{|\mathrm{OPT}|, \frac{1}{\varepsilon}\}$ most valuable items of OPT. Recall, that we break ties by picking first smaller items and then by index, producing a uniquely determined ordering. Denote by $V_{\ell_{\max}}$ and $V_{\ell_{\min}}$ the highest and lowest value class of an element in $\mathrm{OPT}_{\frac{1}{\varepsilon}}$, respectively. Further, we define $n_{\min} \coloneqq |\mathrm{OPT}_{\frac{1}{\varepsilon}} \cap V_{\ell_{\min}}| \leq \frac{1}{\varepsilon}$. Then, items in $V_{\ell_{\min}+1}, \ldots, V_{\ell_{\max}}$ and the first $n_{\min}$ items of $V_{\ell_{\min}}$ can be in $\mathrm{OPT}_{\frac{1}{\varepsilon}}$ but not in $\mathrm{OPT} \setminus \mathrm{OPT}_{\frac{1}{\varepsilon}}$, and the remaining items can be in $\mathrm{OPT} \setminus \mathrm{OPT}_{\frac{1}{\varepsilon}}$ but not in $\mathrm{OPT}_{\frac{1}{\varepsilon}}$. This allows us to approximate $\mathrm{OPT}_{\frac{1}{\varepsilon}}$ and $\mathrm{OPT} \setminus \mathrm{OPT}_{\frac{1}{\varepsilon}}$ with separate subroutines. Denote by v^* the value of the items in $\mathrm{OPT} \setminus \mathrm{OPT}_{\frac{1}{\varepsilon}}$, rounded down to the next power of $(1+\varepsilon)$.

In our algorithms, we will make use of the idea of *guessing* certain properties of an optimal solution. By that we mean that we enumerate all relevant values this property might take and run the subsequent operations for each of them. In the end, we may keep the best solution which will be at least as good as the one in which all guesses are correct. For instance, in the algorithm for knapsack, we will guess the value of $n_{\min}$, so we run the algorithm for all the possible values, i.e., for all values in $\{0, \ldots, \frac{1}{\varepsilon}\}$ (and for each of them possibly multiple times due to other guesses).

To efficiently implement our algorithm, we maintain several instances of the data structure from Lemma 2.1. We store items of each non-empty value class V_ℓ (at most $\log_{1+\varepsilon} \overline{v}$ many) in one instance of the data structure ordered non-decreasingly by size. Second, for each possible value class V_ℓ (at most $\log_{1+\varepsilon} \overline{\overline{v}}$ many), we maintain a data structure instance that contains each input item j with $j \in V_{\ell'}$ for some $\ell' \leq \ell$, ordered non-increasingly by item density $(\frac{v_j}{s_j})$. In particular, we maintain such an instance even if V_ℓ itself is empty (since the data structure might still contain items from classes $V_{\ell'}$ with $\ell' < \ell$). This leads to the additive term in the update time of $O(\log n \log_{1+\varepsilon} \overline{\overline{v}})$. We use additional data structure instances to store the implicit solution and to support queries, as detailed in the algorithm description below.

Algorithm The exact procedure of computing a new implicit solution in an update step, summarized as Algorithm 2.1, is then as follows.

Step 0: Update auxiliary data structures to reflect the insertion or removal of items or knapsacks.

Step 1: Compute a set $H_{\frac{1}{\varepsilon}}$ of high-value candidates. Guess the values of $\ell_{\max}$, $\ell_{\min}$, and $n_{\min}$. If we have $(1+\varepsilon)^{\ell_{\min}} \geq \varepsilon^2 \cdot (1+\varepsilon)^{\ell_{\max}}$, then define $H_{\frac{1}{\varepsilon}}$ to be the set containing the $\frac{1}{\varepsilon}$ smallest items of each of the value classes $V_{\ell_{\min}+1}, \ldots, V_{\ell_{\max}}$, plus the $n_{\min}$ smallest items from $V_{\ell_{\min}}$. Otherwise, set $H_{\frac{1}{\varepsilon}}$ to be the set containing the $\frac{1}{\varepsilon}$ smallest items of each of the value classes with values in $[\varepsilon^2 \cdot (1+\varepsilon)^{\ell_{\max}}, (1+\varepsilon)^{\ell_{\max}}]$.

Step 2: Create a placeholder item B. Guess the value of v^* and consider the auxiliary data structure containing all items with value at most $(1+\varepsilon)^{\ell_{\min}}$ sorted non-increasingly by density. Remove the $n_{\min}$ smallest items of $V_{\ell_{\min}}$ until the end of this iteration/guess. For the remaining items, compute the minimal size of fractional items necessary to reach a value v^* using prefix-sum computation. The placeholder item B is defined to have value $v_B = v^*$ and size s_B equal to the size of those low-value items.

Step 3: Use an FPTAS. On the instance I, consisting of all items in $H_{\frac{1}{\varepsilon}}$ and the placeholder item B, run an FPTAS for static knapsack parameterized by ε (we use the one by Jin [Jin19]) to obtain a packing P.

Step 4: Store implicit solution. Among all guesses, keep the solution P with the highest value. The implicit solution packs the items from $H_{\frac{1}{\varepsilon}}$ as in P and, if $B \in P$, it also packs the low-value items which are completely contained in B (note that at most one item is packed fractionally into B in Step 3 and excluded here). For item queries, store the items in $H_{\frac{1}{\varepsilon}} \cap P$ explicitly. For low-value items, save only the correct guesses of $\ell_{\min}$ and $n_{\min}$, as well as the least dense item contained entirely in B. Membership in B is then recomputed on a query. Also store the value of the solution, computed with the actual, non-rounded item values for possible queries. To do so efficiently, we retrieve the actual item values from the appropriate data structures. We then add the actual values of the packed items from $H_{\frac{1}{\varepsilon}}$ and determining the actual value of items in B with a prefix computation. On a query, we simply return the stored value.

Algorithm 2.1 Dynamic algorithm for a single knapsack

Step 0: Update data structures to reflect insertion or removal of items or knapsacks

Step 1: Compute high-value candidates

1: **for all** guesses of $\ell_{\max}$, $\ell_{\min}$, and $n_{\min}$ **do**

2: $H_{\frac{1}{\varepsilon}} \leftarrow$ items with value at least $\varepsilon^2 \cdot (1+\varepsilon)^{\ell_{\max}}$ that belong to the $\frac{1}{\varepsilon}$ smallest items from $V_{\ell_{\min}+1}, \ldots, V_{\ell_{\max}}$ or the $n_{\min}$ smallest from $V_{\ell_{\min}}$

Step 2: Create placeholder B

3: **for all** guesses of v^* **do**

4: consider density-sorted data structure of items with value $\leq (1+\varepsilon)^{\ell_{\min}}$

5: remove $n_{\min}$ smallest items of $V_{\ell_{\min}}$ temporarily

6: determine s_B via prefix computation of fractional items such that $v_B = v^*$

Step 3: Use FPTAS on $H_{\frac{1}{\varepsilon}} \cup \{B\}$

Step 4: Store solution of highest value implicitly

Queries With the data stored in Step 4, answering queries is simple:

Single-Item Query: If the queried item is contained in $H_{\frac{1}{\varepsilon}}$, its packing was saved explicitly. Otherwise, if B is packed, we saved the last, i.e., least dense, item contained entirely in B. By comparing with this item and considering whether the queried item was removed as one of the $n_{\min}$ smallest items of $V_{\ell_{\min}}$, membership in B can be decided in constant time on a query.

Solution-Value Query: Output the solution value as stored during update.

Query Entire Solution: Output the stored packing of candidates. If the placeholder B was packed, iterate over items in B in the respective density-sorted data structure and output them.

Analysis On an intuitive level, the proof of Theorem 2.5 can be easily summarized. To see that Algorithm 2.1 attains an approximation ratio of $(1-4\varepsilon)$, consider the following steps, each of which contributes a loss of a factor of $(1-\varepsilon)$. (i) A factor of $(1-\varepsilon)$ is lost due to the approximation ratio of the FPTAS. (ii) To obtain a candidate set $H_{\frac{1}{\varepsilon}}$ of cardinality at most $O(\frac{1}{\varepsilon^3})$, we restrict the item values to $[\varepsilon^2 \cdot (1+\varepsilon)^{\ell_{\max}}, (1+\varepsilon)^{\ell_{\max}}]$. Since $|\mathrm{OPT}_{\frac{1}{\varepsilon}}| = \frac{1}{\varepsilon}$, this excludes items from OPT with a total value of at most $\frac{1}{\varepsilon} \cdot \varepsilon^2 (1+\varepsilon)^{\ell_{\max}} \leq \varepsilon \cdot \mathrm{OPT}$. (iii) Due to guessing v^* up to a power of $(1+\varepsilon)$, we get a value of $v_B \geq \frac{1}{1+\varepsilon} \cdot v(\mathrm{OPT} \setminus \mathrm{OPT}_{\frac{1}{\varepsilon}}) \geq (1-\varepsilon) \cdot v(\mathrm{OPT} \setminus \mathrm{OPT}_{\frac{1}{\varepsilon}})$. (iv) Finally, in Step 2, at most one item was packed fractionally. It is charged to the $\frac{1}{\varepsilon}$ items of $\mathrm{OPT}_{\frac{1}{\varepsilon}}$, each of which has a larger value. The running time, too,

can be easily verified by multiplying the numbers of respective guesses for each value as well as the running time of the FTPAS. The latter is $O(\frac{1}{\varepsilon^4})$, since we designed $H_{\frac{1}{\varepsilon}}$ to contain only a constant number of items, namely, at most $O(\frac{1}{\varepsilon^3})$ many.

For the sake of completeness, we provide a formal proof of Theorem 2.5 below. Note, however, that this proof is almost identical to that of Theorem 2.11. It differs only in a few places that correspond to the alterations in Algorithm 2.1 which were made for the setting of multiple but few knapsacks. The reader may thus wish to skip directly to Section 2.3.

To prove Theorem 2.5, we start by considering the iteration in which all our guesses, i.e., $\ell_{\max}$, $\ell_{\min}$, $n_{\min}$, and v^*, are correct and show that the obtained solution has value of at least $(1-4\varepsilon)\cdot v(\mathrm{OPT})$. Let $\mathcal{P}_1$ be the set of solutions respecting: (i) packed items not in $H_{\frac{1}{\varepsilon}}$ have a value of at most $(1+\varepsilon)^{\ell_{\min}}$ but are not part of the $n_{\min}$ smallest items of the value class $V_{\ell_{\min}}$, and (ii) the total value of these items lies in $[v^*, (1+\varepsilon)v^*)$. Denote by OPT_1 the solution of highest value in $\mathcal{P}_1$.

Lemma 2.6. *For* OPT_1 *defined as above,* $v(\mathrm{OPT}_1) \geq (1-\varepsilon)\cdot v(\mathrm{OPT})$.

Proof. Let OPT^* be the packing obtained from OPT by removing all items belonging to $\mathrm{OPT}_{\frac{1}{\varepsilon}}$ whose value is strictly smaller than $\varepsilon^2\,(1+\varepsilon)^{\ell_{\max}}$. By definition, $\mathrm{OPT}_{\frac{1}{\varepsilon}}$ consists of $\frac{1}{\varepsilon}$ items. Thus, the total value of removed items is at most $\frac{1}{\varepsilon}\cdot\varepsilon^2\,(1+\varepsilon)^{\ell_{\max}} \leq \varepsilon\cdot\mathrm{OPT}$. We show that $\mathrm{OPT}^* \in \mathcal{P}_1$.

Consider some $j \in \mathrm{OPT}_{\frac{1}{\varepsilon}}$ with $v_j \geq \varepsilon^2\cdot(1+\varepsilon)^{\ell_{\max}}$. If $\ell_j = \ell_{\min}$, then $j \in H_{\frac{1}{\varepsilon}}$ by definition of $n_{\min}$ and $\mathrm{OPT}_{\frac{1}{\varepsilon}}$, specifically, due to the tie-breaking rules. Assume now that $\ell_j > \ell_{\min}$ and $j \notin H_{\frac{1}{\varepsilon}}$. Recall that $H_{\frac{1}{\varepsilon}}$ contains the $\frac{1}{\varepsilon}$ smallest items of value v_j, and $|\mathrm{OPT}_{\frac{1}{\varepsilon}}| = \frac{1}{\varepsilon}$. Thus, there exists an item of value v_j, smaller than j, which belongs to $H_{\frac{1}{\varepsilon}}$ but not to $\mathrm{OPT}_{\frac{1}{\varepsilon}}$. Exchanging j for this item contradicts the choice of OPT. Therefore, we have $j \in H_{\frac{1}{\varepsilon}}$ and Condition (i) is satisfied. Condition (ii) follows directly from the definition of v^*, and thus $\mathrm{OPT}^* \in \mathcal{P}_1$, concluding the proof. □

Let OPT_2 be the optimal solution of instance I on which the FPTAS is run in Step 3 of Algorithm 2.1.

Lemma 2.7. *We have* $v(\mathrm{OPT}_2) \geq (1-\varepsilon) \cdot v(\mathrm{OPT}_1)$.

Proof. Consider the fractional solution OPT_1^* for I that is obtained from OPT_1 as follows. Place items from $H_{\frac{1}{\varepsilon}}$ the same as they are placed in OPT_1 and additionally place the placeholder item B. Denote by J_L the set of items packed by OPT_1 that are not in $H_{\frac{1}{\varepsilon}}$, that is, the low-value items. By definition of B, we have $v_B = v^* \geq \frac{1}{1+\varepsilon} \cdot v(J_L) \geq (1-\varepsilon) \cdot v(J_L)$. Further, since B consists of the densest low-value items and $v_B \leq c(J_L)$, it must be the case that $s_B \leq s(J_L)$. Therefore, OPT_1^* is a feasible solution for I and we conclude $v(\mathrm{OPT}_2) \geq v(\mathrm{OPT}_1^*) \geq (1-\varepsilon) \cdot v(\mathrm{OPT}_1)$. □

Lemma 2.8. *For the solution* P_F *of Algorithm 2.1,* $v(P_F) \geq (1-4\varepsilon) \cdot v(\mathrm{OPT})$.

Proof. The solution P_{FPTAS} returned by the FPTAS in Step 3 has a value of at least $(1-\varepsilon) \cdot v(\mathrm{OPT}_2)$. The solution P_F is obtained from P_{FPTAS} by replacing the placeholder item with the corresponding low-value items, except possibly one fractionally cut item j. Since there are $\frac{1}{\varepsilon}$ items in OPT that are of higher value than j, namely the ones in $\mathrm{OPT}_{\frac{1}{\varepsilon}}$, this implies

$$v(P_F) \geq v(P_{\mathrm{FPTAS}}) - \varepsilon \cdot v(\mathrm{OPT}).$$

Using Lemmas 2.6 and 2.7, we obtain

$$\begin{aligned} v(P_F) &\geq v(P_{\mathrm{FPTAS}}) - \varepsilon \cdot v(\mathrm{OPT}) \\ &\geq (1-\varepsilon)^2 \cdot v(\mathrm{OPT}_1) - \varepsilon \cdot v(\mathrm{OPT}) \\ &\geq (1-\varepsilon)^3 \cdot v(\mathrm{OPT}) - \varepsilon \cdot v(\mathrm{OPT}) \\ &\geq (1-4\varepsilon) \cdot v(\mathrm{OPT}), \end{aligned}$$

which proves the lemma. □

This concludes the proof of the approximation ratio of Algorithm 2.1. It remains to bound the update and query times.

Lemma 2.9. *It takes a time of* $O(\frac{1}{\varepsilon^8} \log n \log(n\overline{v}) \log^2 \overline{v} + \frac{1}{\varepsilon} \log \overline{v} \log n)$ *to update the solution with Algorithm 2.1.*

Proof. By Lemma 2.2, the first step, i.e., guessing $\ell_{\max}$ and $\ell_{\min}$ and therefore enumerating all possible values, leads to $O(\frac{1}{\varepsilon^2}\log^2\overline{v})$ iterations. Guessing $n_{\min}$ adds an additional factor of $\frac{1}{\varepsilon}$. In the second step, guessing v^* adds a factor of $O(\frac{1}{\varepsilon}\log(n\overline{v}))$ to the running time, since $v(\mathrm{OPT}\setminus\mathrm{OPT}_{\frac{1}{\varepsilon}}) \leq n\overline{v}$ and v^* is rounded to a power of $(1+\varepsilon)$.

By Lemma 2.1, temporarily removing the $n_{\min} \leq \frac{1}{\varepsilon}$ elements from the corresponding data structure costs a total of $O(\frac{1}{\varepsilon}\log n)$, as does adding back removed items from a previous iteration/guess. Computing the size of B can be done by querying the prefix of value just above v^* in time $O(\log n)$ by Lemma 2.1.

For Step 3, we note that the candidate set $H_{\frac{1}{\varepsilon}}$ spans value classes ranging from values $\varepsilon^2\cdot(1+\varepsilon)^{\ell_{\max}}$ or higher to $(1+\varepsilon)^{\ell_{\max}}$. Since all values are rounded to powers of $(1+\varepsilon)$, we consider at most $\log_{1+\varepsilon}\frac{1}{\varepsilon^2}$ different values here. Hence, $H_{\frac{1}{\varepsilon}}$ is composed of $O(\frac{1}{\varepsilon^3})$ items and the FPTAS by Jin [Jin19, Theorem 2] runs in time $O((\frac{1}{\varepsilon^{9/4}}\frac{1}{\varepsilon^{3/2}}+\frac{1}{\varepsilon^2})/2^{\Omega(\sqrt{\log(1/\varepsilon)})}) = O(\frac{1}{\varepsilon^4})$.

Recall, that we need to maintain one copy of the data structure from Section 2.1 for every existing and one for each possible value class, that is, $O(\frac{1}{\varepsilon}\log\overline{v})$ copies in total. Maintenance of these, i.e., insertion or deletion of an item, takes time $O(\frac{1}{\varepsilon}\log\overline{v}\log n)$ in total by Lemma 2.1. □

Lemma 2.10. *The query times of Algorithm 2.1 are as follows.*

(i) Single-item queries are answered in time $O(1)$.
(ii) Solution-value queries are answered in time $O(1)$.
(iii) Queries of the entire solution P are answered in time $O(|P|)$.

The above lemma follows directly from the description of the query operations since comparisons and iterations over a prefix can be done in constant time and time linear in the length of the prefix, respectively.

Proof of Theorem 2.5. The theorem follows directly from Lemmas 2.8 to 2.10. □

2.3 Few Different Knapsacks

It is not very difficult to extend the approach from Section 2.2 to the case of multiple knapsacks by tuning the properties of the candidate set and placeholder and then, naturally, employing an EPTAS for multiple knapsack instead of an FPTAS for knapsack. While applicable for arbitrarily many knapsacks, the running time of this approach is only reasonable when their number is relatively small, i.e., polylogarithmic in n.

The main difference to Section 2.2 comes from the fact that, in order to reserve space for low-value items, a single placeholder is no longer sufficient. Instead, we utilize several smaller placeholders. Since guessing the size of low-value items for every knapsack would lead to a running time exponential in m, we instead employ a sufficiently large number of placeholder items, namely $\frac{m}{\varepsilon}$ many. This leads to additional changes as there are more fractionally packed items, i.e., one per placeholder. To be able to charge them as before, in Lemma 2.8, we now consider the $\frac{m}{\varepsilon^2}$ most profitable items in OPT. This in turn leads to a larger candidate set of size $\frac{m}{\varepsilon^6}$. Other than that, the algorithm remains unchanged.

Theorem 2.11. *For every $\varepsilon > 0$, there is a dynamic algorithm for the multiple knapsack problem that achieves an approximation factor of $(1-\varepsilon)$ with an update time of $2^{O(\frac{1}{\varepsilon}\cdot\log^4(\frac{1}{\varepsilon}))}\cdot(\frac{m}{\varepsilon}\log(n\overline{v}))^{O(1)}+O(\frac{1}{\varepsilon}\log n\log\overline{v})$. Item queries run in time $O(\log\frac{m}{\varepsilon})$, solution-value queries in time $O(1)$, and queries of one knapsack or the entire solution in time linear in the number of output items.*

While conceptually an easy extension, there are some technical details to be taken into account, so we give a full description and detailed analysis.

Notation and Data Structures Let OPT be the set of items used in an optimal solution and $\mathrm{OPT}_{\frac{m}{\varepsilon^2}}$ the set containing the $\frac{m}{\varepsilon^2}$ most valuable items of OPT. Further, denote by $V_{\ell_{\max}}$ and $V_{\ell_{\min}}$ the highest and lowest value (class) of an element in $\mathrm{OPT}_{\frac{m}{\varepsilon^2}}$, respectively, and by $n_{\min}$ the number of elements of $\mathrm{OPT}_{\frac{m}{\varepsilon^2}}$ with value $(1+\varepsilon)^{\ell_{\min}}$. Let v^* be the total value of the items in $\mathrm{OPT}\setminus\mathrm{OPT}_{\frac{m}{\varepsilon^2}}$, rounded down to a power of $(1+\varepsilon)$. The data structures used are identical to those of Section 2.2, i.e., one copy for each non-empty

value class V_ℓ with items from V_ℓ ordered non-decreasingly by size, and one for each possible value class V_ℓ with items of value at most $(1+\varepsilon)^\ell$ ordered non-increasingly by item density. Again, this is the reason for the additive term of $O(\log n \log_{1+\varepsilon} \overline{\overline{v}})$ in the update time. We use additional data structures to store information about the solution, supporting queries, as detailed below.

Algorithm The exact procedure of computing a new implicit solution in an update step, summarized as Algorithm 2.2, is then as follows.

Step 0: Update auxiliary data structures to reflect insertion or removal of items or knapsacks.

Step 1: Compute high-value candidates $H_{\frac{m}{\varepsilon^2}}$. Guess the values $\ell_{\max}$, $\ell_{\min}$, and $n_{\min}$. If $(1+\varepsilon)^{\ell_{\min}} \cdot m \geq \varepsilon^3 \cdot (1+\varepsilon)^{\ell_{\max}}$, define $H_{\frac{m}{\varepsilon^2}}$ to be the set that contains the $\frac{m}{\varepsilon^2}$ smallest items of each of $V_{\ell_{\min}+1}, \ldots, V_{\ell_{\max}}$, plus the $n_{\min}$ smallest items from $V_{\ell_{\min}}$. Otherwise, we set $H_{\frac{m}{\varepsilon^2}}$ to be the set containing the $\frac{m}{\varepsilon^2}$ smallest items of each of the value classes with values in $[\frac{\varepsilon^3}{m} \cdot (1+\varepsilon)^{\ell_{\max}}, (1+\varepsilon)^{\ell_{\max}}]$.

Step 2: Create bundles of low-value items as placeholders. Guess the value v^* and consider the data structure containing all items of value at most $(1+\varepsilon)^{\ell_{\min}}$ sorted by decreasing density. Remove temporarily the $n_{\min}$ smallest items of value $(1+\varepsilon)^{\ell_{\min}}$. Insert them back into the data structure right before the next iteration. From the remaining items, compute the amount of fractional items necessary to reach a value of v^*. That is, sum the sizes of the densest items until their total value is at least v^* and, if necessary, cut the last item fractionally to obtain a set of items with total value v^*. In the same manner, cut this set again fractionally to obtain bundles $B_1, B_2, \ldots, B_{\frac{m}{\varepsilon}}$ of equal value $\frac{\varepsilon}{m} \cdot v^*$.

Step 3: Use an EPTAS. Consider the instance I consisting of the items in $H_{\frac{m}{\varepsilon^2}}$ and the placeholder bundles $B_1, B_2, \ldots, B_{\frac{m}{\varepsilon}}$. Run the EPTAS designed by Jansen [Jan09; Jan12], parameterized by ε, to obtain a packing P for this multiple knapsack instance.

Step 4: Store Implicit Solution. Among all guesses, keep the feasible solution P with the highest value and remove until the next update

the $n_{\min}$ smallest items of $V_{\ell_{\min}}$ from the data structure that contains items of value $(1+\varepsilon)^{\ell_{\min}}$ and lower. Then, for any knapsack, place into the knapsack items from $H_{\frac{m}{\varepsilon^2}}$ as in P and, if B_k is placed in P into this knapsack, also place the low-value items that constitute B_k, except possibly items that were cut fractionally. While used candidates are stored explicitly, the packing of low-value items is stored implicitly by saving the correct guesses and, for every packed bundle, its packing and the first and last, i.e., most and least dense, item that is contained entirely inside the bundle. These items are stored in a data structure D from Section 2.1 sorted by density first and item index second, as in the data structure that was used to compute the bundles. We also store the packing of these *pivot* items explicitly. Also store the value of the solution, computed with the actual, non-rounded item values for possible queries. To do so efficiently, we retrieve the actual item values from the appropriate data structures. We then add the actual values of the packed items from $H_{\frac{1}{\varepsilon}}$ and determining the actual value of items in B with a prefix computation. On a query, we simply return the stored value.

Queries With the data stored in Step 4, answering queries is simple:

Single-Item Query: If the queried item is contained in $H_{\frac{m}{\varepsilon^2}}$, its packing was saved explicitly. For low-value items, we decide its membership in the packing by comparing its density with the pivot elements of packed bundles saved in D. Using the provided search operation, we determine if the item is contained in a bundle and, if this is the case, return its packing.

Solution-Value Query: Output the solution value as stored during update.

Query Packing of Single Knapsack: Output saved packing of candidates in the queried knapsack and iterate in D over the pivot items packed in this knapsack. For each pair of pivot items constituting a bundle, further iterate over all items between these pivot items in the appropriate density-sorted data structure, i.e., that containing items of value $(1+\varepsilon)^{\ell_{\min}}$ and lower except the $n_{\min}$ smallest items of $V_{\ell_{\min}}$, which were temporarily removed, and output their packing.

Query Entire Solution: Apply the above procedure to all knapsacks.

Algorithm 2.2 Dynamic algorithm for few knapsacks

Step 0: Update data structures to reflect insertion or removal of items or knapsacks

Step 1: Compute high-value candidates

1: **for all** guesses of $\ell_{\max}$, $\ell_{\min}$, and $n_{\min}$ **do**

2: $H_{\frac{m}{\varepsilon^2}} \leftarrow$ items with value at least $\frac{\varepsilon^3}{m}(1+\varepsilon)^{\ell_{\max}}$ that belong to the $\frac{m}{\varepsilon^2}$ smallest items from $V_{\ell_{\min}+1}, \ldots, V_{\ell_{\max}}$ or the $n_{\min}$ smallest from $V_{\ell_{\min}}$

Step 2: Create placeholder bundles

3: **for all** guesses of v^* **do**

4: consider density-sorted data structure of items with value $\leq (1+\varepsilon)^{\ell_{\min}}$

5: remove $n_{\min}$ smallest items of $V_{\ell_{\min}}$ temporarily

6: determine via prefix computation items needed to reach value v^*

7: fractionally cut prefix into $\frac{m}{\varepsilon}$ bundles $B_1, B_2, \ldots, B_{\frac{m}{\varepsilon}}$ of equal value $\frac{\varepsilon}{m}v^*$

Step 3: Use EPTAS on $H_{\frac{1}{\varepsilon}} \cup \{B_1, B_2, \ldots, B_{\frac{m}{\varepsilon}}\}$

Step 4: Store solution of highest value implicitly

8: store explicitly the packed candidates and the actual solution value

9: for each packed bundle, store the most and least dense entirely contained item

Analysis The analysis is along the lines of that of Theorem 2.5 with a few changes made to accommodate the alterations described above.

To prove Theorem 2.11, we, again, start with the analysis of the approximation guarantee and consider the iteration of Algorithm 2.2 in which all the guesses, i.e., $\ell_{\max}$, $\ell_{\min}$, $n_{\min}$, and v^*), are correct in order to show that the obtained solution has a value of at least $(1-6\varepsilon)\cdot v(\mathrm{OPT})$. To this end, we consider intermediate results to analyze the impact of each step. Let $\mathcal{P}_1$ be the set of solutions respecting: (i) items not in $H_{\frac{m}{\varepsilon^2}}$ have a value of at most $(1+\varepsilon)^{\ell_{\min}}$ but are not part of the $n_{\min}$ smallest items of the value class $V_{\ell_{\min}}$, and (ii) the total value of these items lies in $[v^*, (1+\varepsilon)v^*)$. Denote by OPT_1 the solution of highest value in $\mathcal{P}_1$.

Lemma 2.12. *For* OPT_1 *defined as above,* $v(\mathrm{OPT}_1) \geq (1-\varepsilon)\cdot v(\mathrm{OPT})$.

Proof. Let OPT^* be the packing obtained from OPT by removing all items belonging to $\mathrm{OPT}_{\frac{m}{\varepsilon^2}}$ whose value is strictly smaller than $\frac{\varepsilon^3}{m}\cdot(1+\varepsilon)^{\ell_{\max}}$. Note that since $\mathrm{OPT}_{\frac{m}{\varepsilon^2}}$ consists of $\frac{m}{\varepsilon^2}$ items, the total value of removed items is at most $\frac{m}{\varepsilon^2}\cdot\frac{\varepsilon^3}{m}\cdot(1+\varepsilon)^{\ell_{\max}} \leq \varepsilon\cdot\mathrm{OPT}$. We show that $\mathrm{OPT}^* \in \mathcal{P}_1$.

Consider an item j in $\mathrm{OPT}_{\frac{m}{\varepsilon^2}}$ of value $v_j \geq \frac{\varepsilon^3}{m}\cdot(1+\varepsilon)^{\ell_{\max}}$. If $\ell_j = \ell_{\min}$, then we have $j \in H_{\frac{m}{\varepsilon^2}}$ due to the definitions of $n_{\min}$ and $\mathrm{OPT}_{\frac{m}{\varepsilon^2}}$, specifically, due to the tie-breaking rules. Assume now that $\ell_j > \ell_{\min}$ and $j \notin H_{\frac{m}{\varepsilon^2}}$. Recall

that $H_{\frac{m}{\varepsilon^2}}$ contains the $\frac{m}{\varepsilon^2}$ smallest items of value v_j, and $|\mathrm{OPT}_{\frac{m}{\varepsilon^2}}| = \frac{m}{\varepsilon^2}$. Thus, there exists an item of value v_j, smaller than j, which belongs to $H_{\frac{m}{\varepsilon^2}}$ but not to $\mathrm{OPT}_{\frac{m}{\varepsilon^2}}$. Exchanging j for this item contradicts the definition of OPT. Therefore, $j \in H_{\frac{m}{\varepsilon^2}}$ and Condition (i) is satisfied. Condition (ii) follows directly from the definition of v^*, and therefore $\mathrm{OPT}^* \in \mathcal{P}_1$, concluding the proof of the lemma. □

Lemma 2.13. *Let OPT_2 be the optimal solution of the instance I on which the EPTAS is run in Step 3. Then, $v(\mathrm{OPT}_2) \geq (1-2\varepsilon) \cdot v(\mathrm{OPT}_1)$.*

Proof. Consider the fractional solution OPT_1^* for I that is obtained using OPT_1 as follows. First, place items from $H_{\frac{m}{\varepsilon^2}}$ as in OPT_1. Next, consider the placeholder bundles $B_1, B_2, \ldots, B_{\frac{m}{\varepsilon}}$ in any order, and place them fractionally into the remaining space. That is, place remaining bundles in the first non-full knapsack. If a bundle does not fit, fill the current knapsack with a fraction of the bundle until it is full, and place the remaining fraction in the next non-full knapsack using the same process. Finally, discard the fractionally cut bundles.

Denote by J_L the set of items packed by OPT_1 that are not in $H_{\frac{m}{\varepsilon^2}}$, i.e., the low-value items. Since the bundles consist of the densest low-value items and $v^* \leq v(J_L)$, it must be the case that $\sum_{k=1}^{\frac{m}{\varepsilon}} s(B_k) \leq s(J_L)$. Therefore, the bundles (fractionally) fit into the remaining space after placing items from $H_{\frac{m}{\varepsilon^2}}$ and OPT_1^* is a feasible solution for I.

By construction of bundles, $\sum_{k=1}^{m} v(B_k) = v^* \geq \frac{1}{1+\varepsilon} v(J_L) \geq (1-\varepsilon) v(J_L)$. Further, since there are $\frac{m}{\varepsilon}$ bundles of equal value and at most m of them are placed fractionally and discarded, we obtain

$$v(\mathrm{OPT}_1^*) \geq (1-2\varepsilon) \cdot v(\mathrm{OPT}_1),$$

which concludes the proof. □

Lemma 2.14. *For the solution P_F of Algorithm 2.2, $v(P_F) \geq (1-6\varepsilon) \cdot v(\mathrm{OPT})$.*

Proof. The solution P_{EPTAS} returned by the EPTAS in Step 3 has a value of at least $(1-\varepsilon) \cdot v(\mathrm{OPT}_2)$. The solution P_F is obtained from P_{EPTAS} by replacing the placeholder bundles with the corresponding low-value items

with the exception of fractionally cut ones, of which there are at most $\frac{m}{\varepsilon}$. Since there are $\frac{m}{\varepsilon^2}$ items in OPT that are of higher value than these items, namely the ones in $\mathrm{OPT}_{\frac{m}{\varepsilon^2}}$, this implies

$$v(P_F) \geq v(P_{\mathrm{EPTAS}}) - \varepsilon \cdot v(\mathrm{OPT}).$$

Using Lemmas 2.12 and 2.13, we obtain

$$\begin{aligned} v(P_F) &\geq v(P_{\mathrm{EPTAS}}) - \varepsilon \cdot v(\mathrm{OPT}) \\ &\geq (1-2\varepsilon)^2 \cdot v(\mathrm{OPT}_1) - \varepsilon \cdot v(\mathrm{OPT}) \\ &\geq (1-2\varepsilon)^2 \cdot (1-\varepsilon) \cdot v(\mathrm{OPT}) - \varepsilon \cdot v(\mathrm{OPT}) \\ &\geq ((1-4\varepsilon) \cdot (1-\varepsilon) - \varepsilon) \cdot v(\mathrm{OPT}) \\ &\geq (1-6\varepsilon) \cdot v(\mathrm{OPT}), \end{aligned}$$

where the second-to-last inequality follows from Bernoulli's inequality. □

This concludes the proof of the approximation guarantee. Next, we prove the running times, starting with that of an update.

Lemma 2.15. *It takes time* $2^{O(\frac{1}{\varepsilon} \cdot \log^4(\frac{1}{\varepsilon}))} \cdot (\frac{m}{\varepsilon} \log(n\overline{v}))^{O(1)} + O(\frac{1}{\varepsilon} \log \overline{v} \log n)$ *to update the solution with Algorithm 2.2.*

Proof. In the first step, guessing $\ell_{\max}$ and $\ell_{\min}$ leads to $O(\frac{1}{\varepsilon^2} \log^2 \overline{v})$ iterations. Guessing $n_{\min}$ adds an additional factor of $\frac{m}{\varepsilon^2}$. In the second step, guessing v^* leads to $O(\frac{1}{\varepsilon} \log(n\overline{v}))$ additional iterations, so the total factor that is attributed to guessing is in $O(\frac{m}{\varepsilon^5} \log^2 \overline{v} \log(n\overline{v}))$.

Temporarily removing the $n_{\min} \leq \frac{m}{\varepsilon^2}$ elements from the data structure containing all items of value at most $(1+\varepsilon)^{\ell_{\min}}$ sorted by decreasing density costs a total of $O(\frac{m}{\varepsilon^2} \log n)$, as does adding back removed items from a previous iteration. Computing the size of the bundles can be done by querying the prefixes of value just above v^*, so in time $O(\log n)$. Computing cut items of bundles takes time $\frac{m}{\varepsilon} \log n$.

The candidate set $H_{\frac{m}{\varepsilon^2}}$ spans value classes with values at least $\frac{\varepsilon^3}{m} \cdot (1+\varepsilon)^{\ell_{\max}}$ to $(1+\varepsilon)^{\ell_{\max}}$. As the value classes correspond to powers of $(1+\varepsilon)$, this means we consider at most $\log_{1+\varepsilon} \frac{m}{\varepsilon^3}$ different value classes. Since $H_{\frac{m}{\varepsilon^2}}$

contains at most $\frac{m}{\varepsilon^2}$ items from each of them, the total number of items is in $O(\frac{m^2}{\varepsilon^6})$. Thus, in the third step, the EPTAS, used on $O(\frac{m^2}{\varepsilon^6})$ items, runs in time $2^{O(\frac{1}{\varepsilon}\cdot\log^4(\frac{1}{\varepsilon}))} + (\frac{m}{\varepsilon})^{O(1)}$. Together, the above constitutes the first term of the update time.

Recall that we need to maintain one data structure for every existing and one for each possible value class, that is, $O(\frac{1}{\varepsilon}\log\overline{v})$ data structures in total. Maintaining these takes time $O(\frac{1}{\varepsilon}\log\overline{v}\log n)$. □

Finally, the following lemma bounds the query times.

Lemma 2.16. *The query times of Algorithm 2.2 are as follows.*

(i) Single-item queries are answered in time $O(\log\frac{m}{\varepsilon})$.
(ii) Solution-value queries are answered in time $O(1)$.
(iii) Queries of the packing P_i of a single knapsack i take time $O(|P_i|)$.
(iv) Queries of the entire solution P are answered in time $O(|P|)$.

Proof. (i): Since the packing of candidates is stored explicitly, each of the packed candidates can be output in time $O(1)$. The part of the solution corresponding to low-value items is stored implicitly with the preparation done during an update. When a low-value item is queried, use the stored pivot items to determine whether the item is contained in packed bundles and, if so, in which bundle it lies and in which knapsack it is packed. Since there are at most $2\cdot\frac{m}{\varepsilon}$ pivot items, this takes time $O(\log\frac{m}{\varepsilon})$.

(ii): The computations for solution-value queries are done during an update of the instance, with the update clearly dominating the running time. Thus, on a query, the answer can be given in constant time.

(iii): As in (i), the packing of candidate items in P_i can be output in time $O(1)$ for each of the items. For low-value items, we iterate over bundles packed in P_i, using the data structure containing the pivot items and starting with the most dense one that is contained in P_i, and output all items of the corresponding bundles. As the groundwork for this was laid in the update step, specifically, the maintenance of data structures and creation of a data structure for pivot items including all relevant pointers, this procedure, too, is linear in the number of output items.

(iv): We use the approach from (iii) on all knapsacks. □

Proof of Theorem 2.11. Lemmas 2.14 to 2.16 together imply the validity of Theorem 2.11. □

2.4 Oblivious Linear Grouping and Multiple Knapsack with Resource Augmentation

In this section, we describe a technique we call *oblivious linear grouping* that is indispensable for us when computing the packing for an unbounded number of items and knapsacks, respectively. Additionally, we give an overview of how this technique may be applied to the multiple knapsack problem when having the advantage of resource augmentation, specifically, when allowing an algorithm (but not the optimal solution its performance is measured against) to use $(\frac{\log n}{\varepsilon})^{O(\frac{1}{\varepsilon})}$ additional knapsacks whose size is equal to that of the largest knapsack of the original instance. For technical details and a complete analysis of these results, we refer the reader to [EMN+20; Ebe20].

2.4.1 Oblivious Linear Grouping

We start with our oblivious linear grouping routine. It specifies a way of rounding the item sizes in order to obtain a small set $\mathcal{T}$ of item types t together with their respective multiplicity n_t. Importantly, it does so in a way that guarantees an optimal solution of the rounded instance to be of almost the same value as an optimal solution of the original instance. We say that two items j, j' are of the same *type* if $j, j' \in V_\ell$ for some ℓ and if their (rounded) sizes are equal. In practice, the rounding of item sizes is done implicitly by computing a set of threshold items and rounding up the size s_j of each item j to that of the next-larger item in this set. Note that we do not have to round each item explicitly since we can use prefix computation to count the number of items between two threshold items. This explains the sublinear running time in the following theorem.

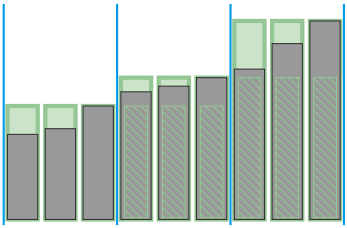


Fig. 2.1: Illustration of linear grouping with 9 items and three groups, indicated by vertical blue lines. Original and rounded items are depicted as gray and green rectangles, respectively. After discarding items of the third group, the rounded items from the first two groups (hatched) fit into the space occupied by original items of the last two groups.

Theorem 2.17. *There exists an algorithm with a running time of* $O(\frac{\log^5 n}{\varepsilon^5})$ *that computes* $O(\frac{\log \overline{v}}{\varepsilon})$ *sets that each contain* $O(\frac{\log^2 n}{\varepsilon^4})$ *item types* t *stored together with their multiplicities* n_t *such that for one of theses sets* $\mathcal{T}$*, the inequality* $v(\mathrm{OPT}_{\mathcal{T}}) \geq (1 - 5\varepsilon) \cdot v(\mathrm{OPT})$ *holds, where* $\mathrm{OPT}_{\mathcal{T}}$ *is an optimal solution for the instance consisting of the rounded items. We can directly compute the set* $\mathcal{T}$*, when given* $\ell_{\max} = \max\{\ell \in \mathbb{Z} \mid V_\ell \cap \mathrm{OPT} \neq \emptyset\}$.

As the name suggests, our oblivious linear grouping routine is closely related to the *linear grouping* routine developed by de la Vega and Luecker [VL81] in the context of bin packing. Roughly speaking, their idea is to solve a bin packing instance with only relatively large items by non-decreasingly sorting items by size and then dividing this range into $O(\frac{1}{\varepsilon^2})$ groups with equal cardinality, except for the first, possibly smaller group. The group with the largest-size items is discarded and the size of any remaining item is then rounded up to the size of the largest item of its group, resulting in only $O(\frac{1}{\varepsilon^2})$ different item sizes overall, so an optimal solution OPT' for this new instance can be computed more efficiently by exploiting the small number of item types.

Indeed, OPT' uses at most as many bins as an optimal solution OPT to the original instance since one could replace the non-rounded items of OPT with a rounded item of the next-smaller group to obtain a solution for the rounded instance that uses at most as many bins as OPT, see Figure 2.1. The previously discarded items of the largest group are packed separately, each in an individual bin. Since the instance consists of relatively large items,

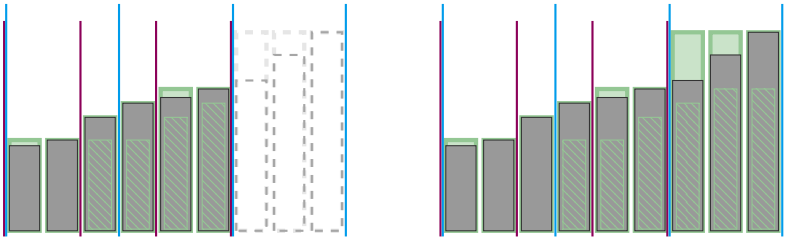


Fig. 2.2: Oblivious linear grouping with two guesses of n_ℓ and three groups, indicated by shorter red and longer blue vertical lines for $n_\ell = 6$ and $n_\ell = 9$, respectively. Rounding of item sizes is done for both guesses simultaneously. The left (right) picture shows the linear grouping of this rounding when the guess $n_\ell = 6$ ($n_\ell = 9$) is correct.

at most $O(\frac{1}{\varepsilon})$ items can fit in any single bin, so the packing of the largest group uses at most $\varepsilon \cdot \mathrm{OPT}$ additional bins.

Returning to the multiple knapsack problem, consider a value class V_ℓ and define n_ℓ to be the number of items of V_ℓ contained in a fixed optimal solution OPT. This value exactly determines which items of V_ℓ are packed in OPT since we may assume that only the smallest items are packed. Knowing the value of n_ℓ would thus allow us to use the linear grouping approach on those first items of V_ℓ, which we know to be in OPT, to reduce the number of different items and then compute a solution more easily. However, in the dynamic setting, guessing the values n_ℓ for all value classes is too time consuming, so we adapt the approach with our oblivious linear grouping routine. Specifically, we determine several relevant guesses of n_ℓ and consider all of them simultaneously while grouping the items. Thereby, the grouping is such that, for each relevant n_ℓ, any group created by oblivious linear grouping must be entirely contained in some group created by the linear grouping of the first n_ℓ elements of V_ℓ, see Figure 2.2 for an example.

Algorithm We now formally describe the rounding procedure satisfying the conditions of Theorem 2.17. It is summarized below as Algorithm 2.3.

Step 1: Determine relevant value classes and guesses for n_ℓ. Guess $\ell_{\max}$ which denotes the largest integer ℓ such that $V_\ell \cap \mathrm{OPT} \neq \emptyset$. Further, let $\ell_{\min} := \ell_{\max} - \left\lceil \log_{1+\varepsilon}(\frac{n}{\varepsilon}) \right\rceil$. We denote by N_ℓ the set of relevant guesses

for n_ℓ, i.e., the set $\{\frac{1}{\varepsilon}\} \cup \{(1+\varepsilon)^k \mid k \in \mathbb{Z} \text{ and } 0 \leq k \leq \log_{1+\varepsilon} n\}$. Do the following for each value class V_ℓ with $\ell_{\min} \leq \ell \leq \ell_{\max}$.

Step 2: Apply linear grouping for each relevant guess. For each $n_\ell \in N_\ell$, consider the n_ℓ smallest elements of V_ℓ. Partition this range of items into $\frac{1}{\varepsilon}$ (almost) equal-sized consecutive groups $G_1(n_\ell), \ldots, G_{\frac{1}{\varepsilon}}(n_\ell)$, consisting of $\lceil \varepsilon n_\ell \rceil$ or $\lfloor \varepsilon n_\ell \rfloor$ elements each, such that $|G_h(n_\ell)| \leq |G_{h'}(n_\ell)|$ for $h \leq h'$. For every group, find the largest item belonging to it, i.e., the element with the highest index in the data structure D_ℓ of items in V_ℓ sorted non-decreasingly by size. Collect them in a set J_ℓ.

Step 3: Combine linear groupings obliviously. Use the items in J_ℓ for each $n_\ell \in N_\ell$ to define the borders of the oblivious linear grouping of V_ℓ and apply the corresponding rounding. That is, we discard all but the smallest $\max N_\ell$ items of V_ℓ and for each remaining item that is not in J_ℓ, we round up its size to that of next item in D_ℓ that is contained in J_ℓ, i.e., the next largest. Add the resulting item types to the set $\mathcal{T}_{\ell_{\max}}$, including those of items in J_ℓ.

Step 4: Output possible type sets. Note that a different set $\mathcal{T}_{\ell_{\max}}$ is computed for each guess of $\ell_{\max}$. Only if we were given the value of $\ell_{\max}$, we could skip the guessing and directly output $\mathcal{T}$ as described in Theorem 2.17. Otherwise, we output all the sets $\mathcal{T}_{\ell_{\max}}$. Then, any algorithms using oblivious linear grouping needs to be executed on the different sets of item types to determine the correct choice.

Analysis While our algorithmic approach is new in the respect that it applies linear grouping simultaneously to many possible values of n_ℓ, the analysis builds on standard techniques. The loss in the objective function due to rounding item values is bounded by a factor of $\frac{1}{1+\varepsilon}$ by Lemma 2.2. As $\ell_{\min}$ is chosen such that n items of value less than $(1+\varepsilon)^{\ell_{\min}}$ contribute less than an ε-fraction of OPT, the loss in the objective function by discarding items in value classes V_ℓ with $\ell < \ell_{\min}$ is bounded by a factor $(1-\varepsilon)$. By taking only $(1+\varepsilon)^{\lfloor \log_{1+\varepsilon} n_\ell \rfloor}$ items of V'_ℓ instead of n_ℓ, we lose at most a factor $\frac{1}{1+\varepsilon}$. The groups created by oblivious linear grouping are an actual refinement of the groups created by classical linear grouping. Thus, we pack our items similarly: not packing the group with the largest items (at the loss of a factor

Algorithm 2.3 Oblivious linear grouping

Step 1: Determine relevant value classes and guesses for n_ℓ

for all guesses of $\ell_{\max} = \max\{\ell \mid V_\ell \cap \mathrm{OPT} \neq \emptyset\}$ **do**
  $\ell_{\min} \leftarrow \ell_{\max} - \lceil \log_{1+\varepsilon} \frac{n}{\varepsilon} \rceil$
  for all ℓ with $\ell_{\min} \leq \ell \leq \ell_{\max}$ **do**

    Step 2: Apply linear grouping for each relevant guess
    $J_\ell \leftarrow \emptyset$
    for all $n_\ell \in N_\ell := \{\frac{1}{\varepsilon}\} \cup \{(1+\varepsilon)^k \mid 0 \leq k \leq \log_{1+\varepsilon} n\}$ **do**
      partition smallest n_ℓ items of V_ℓ into $\frac{1}{\varepsilon}$ equal-size groups
      add largest item of each group to J_ℓ

    Step 3: Combine linear groupings obliviously
    round up size of the $\max N_\ell$ smallest items in V_ℓ to that of the next item in D_ℓ that is also in J_ℓ, and add resulting item types to $\mathcal{T}_{\ell_{\max}}$

Step 4: Output all possible type sets $\mathcal{T}_{\ell_{\max}}$.

of $(1-2\varepsilon)$) allows us to "move" all rounded items of group $G_k(n_\ell)$ to the positions of the (not rounded) items in group $G_{k+1}(n_\ell)$. Combining the above, we obtain $v(\mathrm{OPT}_{\mathcal{T}}) \geq \frac{(1-\varepsilon)(1-2\varepsilon)}{(1+\varepsilon)^2} v(\mathrm{OPT})$.

Since $\mathcal{T}$ contains at most $\frac{1}{\varepsilon}(\lceil \frac{\log \frac{n}{\varepsilon}}{\log(1+\varepsilon)} \rceil + 1)$ different value classes, and as it suffices to use $\lceil \frac{\log n}{\log(1+\varepsilon)} \rceil + 1$ different values for $n_\ell = |\mathrm{OPT} \cap V_\ell|$, we have $|\mathcal{T}| \leq O(\frac{\log^2 n}{\varepsilon^4})$. Using the access times given in Lemma 2.1 bounds the running time. For detailed and complete proofs, see [EMN+20; Ebe20].

2.4.2 Multiple Knapsacks with Resource Augmentation

We consider instances for multiple knapsack with arbitrarily many knapsacks of different capacities and show how to efficiently maintain a $(1-\varepsilon)$-approximation when given, as resource augmentation, $A \in (\frac{\log n}{\varepsilon})^{O(\frac{1}{\varepsilon})}$ additional knapsacks that have the same capacity as a largest knapsack in the input instance. While we may pack items into these additional knapsacks, the optimal solution used to measure our algorithm's performance against is not allowed to do so. Recall that $S_{\max}$ is defined as the currently largest capacity of any knapsack.

Theorem 2.18. *For every $\varepsilon > 0$, there is a dynamic algorithm for multiple knapsack that, given A additional knapsacks as resource augmentation,*

for some $A \in (\frac{\log n}{\varepsilon})^{O(\frac{1}{\varepsilon})}$*, maintains* $(1-\varepsilon)$*-approximate implicit solutions. It does so with an update time of* $(\frac{\log n}{\varepsilon})^{O(\frac{1}{\varepsilon})}(\log S_{\max} \log \overline{v})^{O(1)}$*. Item queries take time* $O(\log(\frac{\log n}{\varepsilon}))$*, and the entire solution* P *can be output in time* $O(\log(\frac{\log n}{\varepsilon})|P|)$*.*

The algorithm crucially relies on the oblivious linear grouping routine to reduce the number of different item types, which then allows the use of a configuration ILP. A configuration describes a different way of packing available item types and is specific to the type of knapsack in which it can be packed. More precisely, knapsacks are partitioned into a minimal number of groups such that any item type is considered to be either *big* or *small* with respect to every knapsack in the group, meaning that its size is at least ε times the knapsack capacity or less than that, respectively. A *configuration* for a knapsack group is then a multiset containing only item types that are big with respect to a knapsack in the group. Additionally, at most n_t items of type $t \in \mathcal{T}$ are contained in any single configuration and the total size of items in a configuration does not exceed the size of the largest knapsack in the group. It follows that any configuration contains at most $\frac{1}{\varepsilon}$ items. Small items are considered separately by the ILP and space for them is reserved. After solving the LP relaxation of the configuration ILP, the obtained solution is rounded to an integral packing and small items are packed fractionally in the remaining space. Here, the resource augmentation is needed to ensure sufficient capacity even after rounding the configuration variables. The LP formulation is similar in spirit to that in [Jan09; Jan12]; however, we only use configurations of big items and have a polylogarithmic number of item types, yielding a smaller LP that can be solved faster.

Notation and Data Structures We assume again that item values are rounded to powers of $(1+\varepsilon)$, which results in value classes V_ℓ of items with value $v_j = (1+\varepsilon)^\ell$. An item j is said to be *big* with respect to a knapsack i if we have $s_j \geq \varepsilon \cdot S_i$, and *small* otherwise. We maintain three different balanced kinds of binary trees as described in Section 2.1. 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 data structure instance where the items are sorted non-decreasingly by time of arrival. For each value class V_ℓ, we maintain one data structure instance for sorting the items with $\ell_j = \ell$ ordered non-decreasingly

in size. We also store the knapsacks sorted non-increasingly by capacity in an additional instance of the data structure.

Algorithm We assume that $m \geq \frac{16}{\varepsilon^7} \log n$, as otherwise we may use the Algorithm 2.2 from Section 2.3. The procedure of computing an implicit solution in an update step, summarized as Algorithm 2.4, is as follows.

Step 0: Update auxiliary data structures to reflect the insertion or removal of items or knapsacks.

Step 1: Apply oblivious linear grouping. Guess $\ell_{\max}$, the index of the highest value class that belongs to OPT and use oblivious linear grouping to obtain $\mathcal{T}$, the set of item types t with their multiplicities n_t.

Step 2: Group knapsacks. Consider the knapsacks sorted non-decreasingly by their capacity and determine for each item size for which knapsacks a corresponding item would be considered big or small. This yields a set $\mathcal{G}$ of $O(\frac{\log^2 n}{\varepsilon^4})$ knapsack groups, see Figure 2.3a. Denote by $\mathcal{F}_g$ the set of item types that are small with respect to a group $g \in \mathcal{G}$, by S_g the total capacity of all knapsacks in g, and by $S_{g,\varepsilon}$ the total capacity of the smallest $\lceil \varepsilon m_g \rceil$ knapsacks in g. Further, denote by m_g the number of knapsacks in group g and by $\mathcal{G}_{\frac{1}{\varepsilon}}$ the subset of groups $g \in \mathcal{G}$ with $m_g \geq \frac{1}{\varepsilon}$.

Step 3: Create configurations of big items. For each group $g \in \mathcal{G}$, create all feasible configurations consisting of at most $\frac{1}{\varepsilon}$ items that are big with respect to knapsacks in g. This amounts to $O((\frac{\log^2 n}{\varepsilon^4})^{\frac{1}{\varepsilon}})$ configurations per group. Order the configurations non-increasingly by total item size and denote the set of such configurations as $\mathcal{C}_g = \{c_{g,1}, c_{g,2}, \ldots, c_{g,k_g}\}$. Note that, by this definition, we may have duplicates of the same configuration for different groups (but we view them as distinct items). Let $m_{g,\ell}$ be the number of knapsacks in group g that have sufficient capacity to contain 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.

Step 4: Solve configuration ILP fractionally. Solve the following configuration ILP (P) with variables y_c and $z_{g,t}$ for $c \in \mathcal{C}$, $g \in \mathcal{G}$, and $t \in \mathcal{T}$. The variables y_c count how often $c \in \mathcal{C}$ is used, i.e., the number of occurrences of (specific duplicates of) a configuration, and variables $z_{g,t}$ count, for item

types t that are considered small with respect to g, how many items of type t are packed in knapsacks of group g.

$$
\begin{aligned}
\max \quad & \sum_{g\in\mathcal{G}}\sum_{c\in\mathcal{C}_g} y_c v_c + \sum_{g\in\mathcal{G}}\sum_{t\in\mathcal{F}_g} z_{g,t} v_t & & & \\
\text{s.t.} \quad & \sum_{h=1}^{\ell} y_{c_{g,h}} & \leq\ & m_{g,\ell} & \text{for all } g\in\mathcal{G}, \ell\in[k_g] \\
& \sum_{c\in\mathcal{C}_g} y_c & \leq\ & (1-\varepsilon)m_g & \text{for all } g\in\mathcal{G}_{\frac{1}{\varepsilon}} \\
& \sum_{c\in\mathcal{C}_g} y_c s_c + \sum_{t\in\mathcal{F}_g} z_{g,t} s_t & \leq\ & S_g & \text{for all } g\in\mathcal{G}\setminus\mathcal{G}_{\frac{1}{\varepsilon}} \\
& \sum_{c\in\mathcal{C}_g} y_c s_c + \sum_{t\in\mathcal{F}_g} z_{g,t} s_t & \leq\ & S_g - S_{g,\varepsilon} & \text{for all } g\in\mathcal{G}_{\frac{1}{\varepsilon}} \\
& \sum_{g\in\mathcal{G}}\sum_{c\in\mathcal{C}_g} y_c n_{c,t} + \sum_{g\in\mathcal{G}:t\in\mathcal{F}_g} z_{g,t} & \leq\ & n_t & \text{for all } t\in\mathcal{T} \\
& z_{g,t} & =\ & 0 & \text{for all } g\in\mathcal{G}, t\in\mathcal{T}\setminus\mathcal{F}_g \\
& z_{g,t} & \in\ & \mathbb{Z}_{\geq 0} & \text{for all } g\in\mathcal{G}, t\in\mathcal{T} \\
& y_c & \in\ & \mathbb{Z}_{\geq 0} & \text{for all } g\in\mathcal{G}, c\in\mathcal{C}_g
\end{aligned}
\tag{P}
$$

The first inequality ensures that the configurations chosen by the ILP actually fit into the knapsacks of the respective group. In particular, it prevents configurations that are too large for a knapsack to be packed into it fractionally. The second inequality ensures that an ε-fraction of knapsacks in $\mathcal{G}_{\frac{1}{\varepsilon}}$ remains empty for packing small fractionally 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. We relax the ILP by allowing $y_c, z_{g,t} \in \mathbb{R}_{\geq 0}$ and solve the resulting LP to obtain a basic feasible optimal solution.

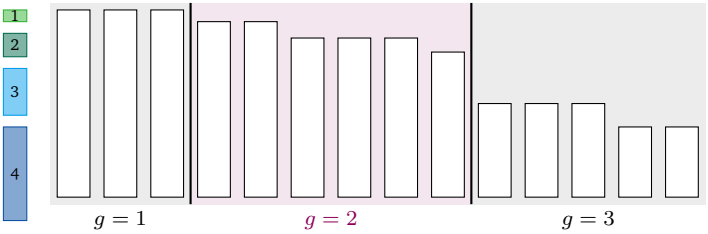
Step 5: Round the LP solution. For each non-zero variable of the basic feasible solution, round down its value to the next integer. Whenever a variable is rounded in this manner (that is, when its value was not an integer before the rounding) place one corresponding element, i.e., item of a certain type or a configuration, into one of the knapsacks which is granted by the resource augmentation. We place configurations within a group such that

knapsacks containing the same configuration form a contiguous range in the data structure in which they are stored.

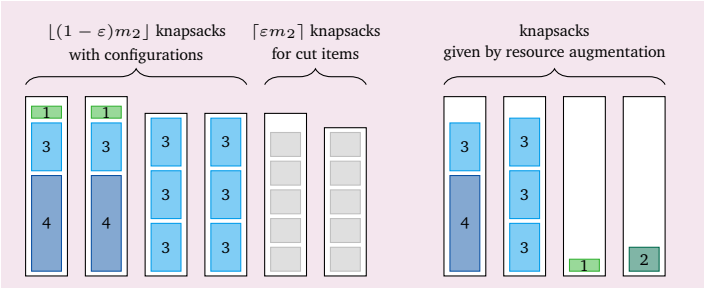
Note that, since we are using item types and implicit solutions, placing an item merely means that we reserve a spot for an item of this type, which can be filled on a query. In fact, with this rounding procedure, it might be that we reserve more item spots than there are items of this type, which needs to be taken into account when computing the solution value but is not a problem otherwise as the spot simply is not used. When computing the solution value, we simply compare the number of reserved spots for a certain type with the number of items actually available.

Step 6: Pack the small items. Observe that the variables $z_{g,t}$, even after rounding, pack the small item types $t \in \mathcal{F}_g$ fractionally since they are only assigned to a knapsack group as a whole. We can obtain a final solution by packing small items, for each group $g \in \mathcal{G}_{\frac{1}{\varepsilon}}$, in a next-fit-like manner in the remaining space left after packing the big items (except those small items already placed in a resource-augmentation knapsack during the previous step). Consider an arbitrary order of small items and knapsacks allocated for big items (i.e., not the $(1-\varepsilon)m_{\mathcal{G}}$ reserved knapsacks), respectively. The exact order used for the items will be determined by the order in which they are queried. Start filling small items into the first knapsack in this order until one does not fit. Pack this item instead into the next empty slot of one of the εm_g additional knapsacks reserved in the second inequality of (P) (possibly opening a new one) and call such an item cut. Consider the next knapsack in the order and continue filling small items in this manner. For groups $g \in \mathcal{G} \setminus \mathcal{G}_{\frac{1}{\varepsilon}}$, we use the same approach, but the cut items are placed in a knapsack granted by the resource augmentation.

Step 7: Store information for queries and fix guess of $\ell_{\max}$. Fix the guess of $\ell_{\max}$ attaining the highest value. To prepare for answering queries, save which and how many configurations are packed into which knapsacks. Since within a group instances of the same configuration are stored contiguously we can do that implicitly by only storing the first and last relevant knapsack. For each item type t, we maintain an ordered list that stores knapsack groups g which contain an item of type t together with the information how many items of type t are to be packed in this group. Further, for each such group g for which t is considered a big item, we maintain a list of configurations



(a) Knapsack groups from item types.



(b) Possible solution for group 2.

Fig. 2.3: Subfigure (a) shows three knapsack groups. Item types 2, 3, and 4, are big with respect to knapsacks in groups with index at least 3, 2, and 1, respectively. Items of type 1 are small with respect to all knapsacks. Subfigure (b) shows a possible solution of Algorithm 2.4 for group 2. Space for cut small items is reserved (gray) by the linear program so they fit into the original knapsacks. Resource augmentation is used only for rounding the fractional solution. Due to their smaller cardinality, groups 1 and 3 instead use additional knapsacks from resource augmentation to pack cut items.

that contain an item of type t and for each entry store the amount of such configurations packed in g, as well as a pointer to the first knapsack containing one of its instances. These lists will determine into which knapsack the *next* item of type t will be packed upon its first query by iterating through the lists, updating pointers, and placing as many items as appropriate while keeping track of the progress. Lastly, we use prefix computation to determine the set of packed items and compute and store the not-rounded solution value. For each value class, we store the largest item that is contained in the solution.

A possible solution from Algorithm 2.4 is shown in Figure 2.3.

Algorithm 2.4 Dynamic algorithm for multiple knapsack with resource augmentation

Step 0: Update data structures to reflect insertion or removal of items or knapsacks

Step 1: Apply oblivious linear grouping and guess $\ell_{\max}$

Step 2: Group knapsacks

1: $\mathcal{G} \leftarrow$ set of $O(\frac{\log^2 n}{\varepsilon^4})$ knapsack groups such that a fixed item is either small or big with respect to all knapsacks of the same group

Step 3: Create configurations of big items

2: **for all** $g \in \mathcal{G}$ **do**

3: $\mathcal{C}_g \leftarrow$ set of all configurations containing at most $\frac{1}{\varepsilon}$ items that are big with respect to g. We have $|\mathcal{C}_g| = O((\frac{\log^2 n}{\varepsilon^4})^{\frac{1}{\varepsilon}})$

Step 4: Solve the configuration ILP (P) fractionally

Step 5: Round the LP solution

4: round value of every non-zero variable in solution down to the nearest integer

5: place into an individual knapsack from the resource augmentation one element for each rounded variable

Step 6: Pack small items

6: pack small items next-fit-like in free space of knapsacks with big items

7: place cut items into knapsacks reserved by (P) or from resource-augmentation

Step 7: Store information for queries and fix guess of $\ell_{\max}$

Queries Using the information stored in Step 7, the query operations work as follows.

Single-Item Query: If the item was queried already since the last update, output the packing stored during that query. Compare the item to the stored element of its value class, i.e., the largest item of the class contained in the solution, as specified in Step 7, to find out if the queried item was packed. If not, output this information. Otherwise, determine the item type t (in time $O(\log(\frac{\log n}{\varepsilon}))$) and the group g into which the next item of type t is packed. If t is considered small with respect to g, pack it into the empty space not reserved for configurations or, if the item turns out to be a cut item, into an additional knapsack (reserved by (P) or from resource augmentation) as detailed in Step 6 of the algorithm. If t is considered big, pack it into the next knapsack as determined by the lists stored in Step 7.

Store the packing of this item explicitly in case of a repeated query.

Solution-Value Query: Return the solution value as stored in Step 7.

Query Entire Solution: Iterate over all items stored in the solution, using the stored largest elements of each value class, and use the single-item query on each item to determine its knapsack.

Analysis For the full analysis, we again refer to [EMN+20; Ebe20]. We only briefly hint at why the number A of added knapsacks is sufficient to accommodate cut items and the additional items and configurations from rounding the LP solution. Recall that $|\mathcal{G}|, |\mathcal{T}| \in O(\frac{\log^2 n}{\varepsilon^4})$, and $|\mathcal{C}_g \mid \in (\frac{\log n}{\varepsilon})^{O(\frac{1}{\varepsilon})}$ for every group $g \in \mathcal{G}$. Thus, the ILP (P) has at most $(\frac{\log n}{\varepsilon})^{O(\frac{1}{\varepsilon})}$ variables, which is an upper bound on the number of additional knapsacks needed during rounding. Further, by the definition of small items, all cut items from a knapsack group $g \in \mathcal{G} \setminus \mathcal{G}_{\frac{1}{\varepsilon}}$ fit into a single additional knapsack, adding another $O(\frac{\log^2 n}{\varepsilon^4})$ term to the necessary number of additional knapsacks. So clearly, $A \in (\frac{\log n}{\varepsilon})^{O(\frac{1}{\varepsilon})}$ suffices.

2.5 Solving Multiple Knapsack

Finally, after having laid the groundwork in the two previous sections, we show how to maintain solutions for arbitrary instances of the multiple knapsack problem. This is the main result of the chapter, summarized in the following theorem.

Theorem 2.19. *For every $\varepsilon > 0$, there is a dynamic algorithm for the multiple knapsack problem that maintains $(1-\varepsilon)$-approximate implicit solutions with update time $(\frac{\log n}{\varepsilon})^{O(\frac{1}{\varepsilon})}(\log \overline{v} \cdot \log S_{\max})^{O(1)}$. Item queries are served in time $O(\log(\frac{\log n}{\varepsilon}))$ and the solution P can be output in time $O(\log(\frac{\log n}{\varepsilon})|P|)$, its value in time $O(1)$.*

The intuition here is as follows. We would like to use Algorithm 2.4 on the multiple knapsack instance but without actually using resource augmentation. To achieve this, we partitioning the knapsacks into three sets of non-increasing capacities, whose knapsacks we shall call *special*, *extra*, and *ordinary*. We apply Algorithm 2.4 only to the ordinary knapsacks, with a portion of the extra knapsacks acting as the additional knapsacks for "resource augmentation". The special knapsacks are the $(\log n)^{O(\frac{1}{\varepsilon})}$ largest input knapsacks and we

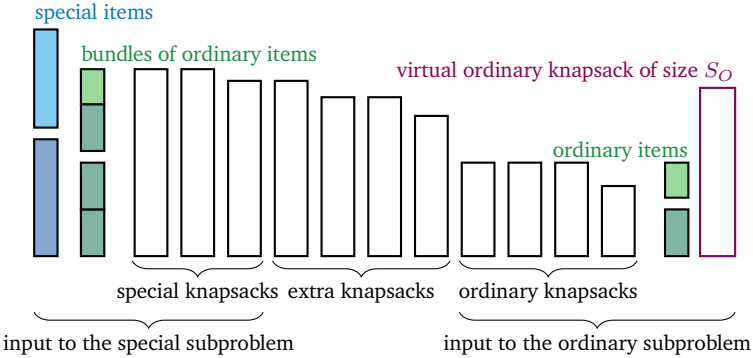


Fig. 2.4: Input of the special and the ordinary subproblems: Based on the current guess for the extra knapsacks, the knapsacks are partitioned into three groups (special, extra, and ordinary). When an item fits into at least one ordinary knapsack, it is ordinary and special otherwise. The total size of ordinary items placed by OPT in special knapsacks gives the size of the virtual ordinary knapsack. The ordinary items packed into this virtual knapsack are further assigned to bundles of equal size, which are then part of the input to the special subproblem.

aim to apply Algorithm 2.2 to them (for a suitably defined set of input items). Extra knapsacks are $(\log n)^{O(\frac{1}{\varepsilon})}$ knapsacks that are smaller than the special knapsacks, but larger than the ordinary knapsacks. Crucially, we choose the partition of knapsacks in a way that ensures that there is a (global) $(1-\varepsilon)$-approximate solution in which all extra knapsacks are empty, enabling us to use them solely for the purpose of mimicking resource augmentation in the ordinary subproblem. We note that while some similarities to the approach in [Jan09] exist, there, only two groups of knapsacks suffice.

Definitions and Data Structures Denote by $\mathrm{OPT}_{\mathcal{T}}$ a fixed optimal solution of the instance that is obtained after using oblivious linear grouping, i.e., Algorithm 2.3, on the input instance. Choose $A = (\frac{\log n}{\varepsilon})^{\Theta(\frac{1}{\varepsilon})}$. We assume that $m > (\frac{1}{\varepsilon})^{\frac{4}{\varepsilon}} \cdot A$, since otherwise we may simply apply Theorem 2.11. Consider $\frac{1}{\varepsilon}$ groups of knapsacks with sizes $\frac{A}{\varepsilon^{3i}}$, for $i = 0, 1, \ldots, \frac{1}{\varepsilon} - 1$, such that the first group, i.e., $i = 0$, consists of the A largest knapsacks, the second, i.e., $i = 1$, of the $\frac{A}{\varepsilon^3}$ next largest, and so on. In $\mathrm{OPT}_{\mathcal{T}}$, one of these groups contains items with total value at most $\varepsilon \cdot \mathrm{OPT}_{\mathcal{T}}$. Let $k \in \{0, 1, \ldots, \frac{1}{\varepsilon} - 1\}$ be the index of such a group and let $A_S := \sum_{i=0}^{k-1} \frac{A}{\varepsilon^{3i}}$. We say that the A_S

largest knapsacks of the input are the *special* knapsacks. The *extra* knapsacks are the $\frac{A}{\varepsilon^{3k}} > \frac{A_S}{\varepsilon^2} + A$ next largest, and the *ordinary* knapsacks the remaining ones (including those not in one of the $\frac{1}{\varepsilon}$ groups).

Call an item *ordinary* if it fits into the largest ordinary knapsack and *special* otherwise. Denote by J_O and J_S the set of ordinary and special items, respectively, and by S_O the total size of ordinary items that $\mathrm{OPT}_{\mathcal{T}}$ places in special knapsacks, rounded down to the next power of $(1+\varepsilon)$; see Figure 2.4. We use the same data structures as in Algorithm 2.4, which is used as a subroutine. The data structures for Algorithm 2.2, which is also used as a subroutine, are built from scratch for every new guess and update using item types. As we will see, this is not a problem for the running time, since it is possible to bundle items with the same type when first (re-)creating the data structure. Because there are only few item types, this does not impact the overall running time.

Algorithm The detailed steps for updating an implicit solution, summarized in Algorithm 2.5, are as follows.

Step 0: Update auxiliary data structures to reflect the insertion or removal of items or knapsacks.

Step 1: Apply oblivious linear grouping. Guess $\ell_{\max}$, the index of the highest value class that belongs to OPT and use the oblivious linear grouping technique, as described in Section 2.4.1, to obtain $\mathcal{T}$, the set of $O(\frac{\log^2 n}{\varepsilon^4})$ different item types t with their multiplicities n_t. Guess k and determine whether items of a certain type are ordinary or special.

Step 2: Pack high-value ordinary items. Place each of the $\frac{A_S}{\varepsilon^2}$ most valuable ordinary items in an empty extra knapsack and denote this set of items by J_E. If there are not enough red items, we may solve the entire instance using Algorithm 2.2.

Step 3: Add a virtual ordinary knapsack. Guess S_O and add a virtual knapsack with capacity S_O to the instance in the ordinary subproblem.

Step 4: Solve the ordinary subproblem. Remove temporarily the set J_E from the data structures of the ordinary subproblem. Solve the subproblem via Algorithm 2.4 and use A extra knapsacks for resource augmentation. In

the LP, treat every ordinary item as small item with respect to this knapsack and do not use configurations. When rounding up variables, fill the $O(\frac{\log^2 n}{\varepsilon^4})$ rounded items from the virtual knapsack into extra knapsacks.

Step 5: Create bundles from virtual knapsack. Consider the items that remain in the virtual ordinary knapsack after rounding. Sort them by type, highest value first, and cut them fractionally to form a set B_O consisting of $\frac{A_S}{\varepsilon}$ item bundles of equal size. For each bundle, remember how many items of each type are placed entirely inside it. Place cut items into extra knapsacks. Consider each $B \in B_O$ as an item of size and value equal to the combined size and value, respectively, of items placed entirely in B.

Step 6: Solve the special instance. Create the subinstance for the special subproblem and set up the required data structures with item types. Specifically, instead of populating the data structures with individual items, we consider elements that store an item type together with its multiplicity to group together items of one type, facilitating a faster initialization. In this manner, we may still use prefix computation and, when needed, e.g., for cutting bundles, split up such an element into several of these elements with their multiplicities summing up to that of the original one, or one less due to fractional cutting. Only for candidate items, when invoking the EPTAS, this implicit representation is resolved into actual items. Insert the bundles in B_O into the data structures and apply Algorithm 2.2.

Step 7: Store implicit solution. Among all guesses, keep the solution P_F with the highest value. Store items in J_E and their placement explicitly. Revert the removal of J_E from the ordinary data structures at the start of the next update. For the remaining items, the solutions are given as in the respective subproblem, with the exception of items packed in the virtual ordinary knapsack. To support the query of such items, we save the first, i.e., the smallest least valuable, and last item in a bundle of B_O that is packed in the final solution. Membership in the solution can then be decided on a query as described below.

Queries To answer queries, we essentially use the same approaches as in Sections 2.3 and 2.4.2 for the ordinary and special subproblem, respectively, and generally pack the first $\bar{n}_t$ queried items of type t, where $\bar{n}_t$ is the number

Algorithm 2.5 Dynamic algorithm for multiple knapsacks

Step 0: Update data structures to reflect insertion or removal of items or knapsacks

Step 1: Apply oblivious linear grouping

1: **for all** guesses of $\ell_{\max}$ and k **do**

2: use Algorithm 2.3 to obtain item types; determine ordinary and special items

3: **Step 2:** Pack the $\frac{A_S}{\varepsilon^2}$ most valuable ordinary items (J_E) in extra knapsacks

Step 3: Add a virtual ordinary knapsack

4: **for all** guesses of S_O **do**

5: add a virtual knapsack with capacity S_O to the ordinary subproblem, treat all items as small for this knapsack in LP used by Algorithm 2.4

Step 4: Solve the ordinary subproblem

6: remove J_E; solve ordinary subproblem with Algorithm 2.4, use extra knapsacks for resource augmentation and rounding virtual knapsack

Step 5: Create bundles from virtual ordinary knapsack

7: cut items in virtual knapsack fractionally into $\frac{A_S}{\varepsilon}$ bundles of equal size

8: place cut items into extra knapsacks

Step 6: Solve the special instance

9: insert bundles into the special subproblem and solve it with Algorithm 2.2

Step 7: Store implicit solution

10: among all guesses, choose highest-value solution; save information for queries

of items with type t packed in the implicit solution. However, since the special subproblem now also works with item types, we integrate its implicit solution into the lists from Section 2.4.2 that dictate where the "next" item of a certain type is packed. We do this by including the candidate set $H_{\frac{1}{\varepsilon}}$ and the bundles in the list of type t, if an item of type t is packed as a high-value candidate or is contained in a packed bundle, respectively, and store the number of items of type t that are packed in the respective location as well. When an item of type t is queried and the list points to $H_{\frac{1}{\varepsilon}}$ or a bundle as the place where to look next, we use the explicitly stored information in the special subproblem to resolve the query. Since here all bundles have the same value, we may assume that the smallest bundles are packed, and instead of pivot items, we simply save the number of items of a certain type packed in a bundle.

Moreover, special care has to be taken with items that were packed in the virtual knapsack in the ordinary subproblem. Here, we assumed that items of a certain type which are packed in the virtual knapsack are the first, i.e., smallest, of that type. We can therefore decide in constant time whether or not an item is contained in the virtual knapsack. Further, we assume

that in the special subproblem the highest-value bundles in B_O are packed. Thus, we can decide in constant time, whether or not an item of the virtual knapsack is packed in the final solution and then fill it into the free space in special knapsacks reserved by bundles. We do this simply by using a first fit algorithm on the knapsacks with reserved space. Since items in extra knapsacks are stored explicitly, they can be accessed in constant time.

Analysis To prove Theorem 2.19, we consider again the iteration in which all the guesses are correct, namely, the guesses of $\ell_{\max}$, k and S_O. Recall that $\mathrm{OPT}_{\mathcal{T}}$ denotes a fixed optimal solution of the knapsack instance after rounding via oblivious linear grouping in Step 1 for the correct guess of $\ell_{\max}$ and consider this rounded instance from here on. Let $\mathcal{P}_1$ be the set of solutions on the ordinary knapsacks (without the additional virtual knapsack) and the special knapsacks such that the total size of ordinary items placed in special knapsacks lies in the range $[S_O, (1+\varepsilon)S_O)$. We make the following observations about optimal solutions in $\mathcal{P}_1$.

Observation 2.20. *For a solution $\mathrm{OPT}_1 \in \mathcal{P}_1$ that has the highest value, we have $v(\mathrm{OPT}_1) \geq (1-\varepsilon) \cdot v(\mathrm{OPT}_{\mathcal{T}})$.*

This can be easily seen as follows. Altering $\mathrm{OPT}_{\mathcal{T}}$ by deleting the extra knapsacks gives a solution in $\mathcal{P}_1$ of value at least $(1-\varepsilon)\cdot v(\mathrm{OPT}_{\mathcal{T}})$. This holds since, for the correct guess of k, the extra knapsacks by definition contribute at most an ε-fraction to the value of $\mathrm{OPT}_{\mathcal{T}}$. Further, the correctness of the guess of S_O implies that the altered $\mathrm{OPT}_{\mathcal{T}}$ is indeed a packing in $\mathcal{P}_1$.

Lemma 2.21. *Consider an optimal solution OPT_O to the ordinary subproblem, that it, exclude items in J_E but include virtual knapsack. Further, define $\mathrm{OPT}_{1,O} \coloneqq (\mathrm{OPT}_1 \cap J_O) \setminus J_E$. Then $v(\mathrm{OPT}_O) \geq v(\mathrm{OPT}_{1,O}) - 2\varepsilon \cdot v(\mathrm{OPT}_{\mathcal{T}})$.*

Proof. Consider the ordinary items in OPT_1 that are not in J_E. Leave items in ordinary knapsacks in their current position and place ordinary items in special knapsacks into the virtual ordinary knapsack. The latter is possible with the exception of possibly an ε-fraction of the items (with respect to size) due to S_O being rounded down. Deleting the least dense items until the remainder fits into the virtual knapsack causes a loss of at most an ε-fraction

of the value of OPT_1 plus an additional ordinary item j_O. This item j_O contributes at most an ε-fraction to $\mathrm{OPT}_\mathcal{T}$ as its value is not larger than that of the least valuable element in J_E, which has a value of at most $\varepsilon \cdot v(\mathrm{OPT}_\mathcal{T})$. □

Lemma 2.22. *Let P_F denote the final rounded solution computed by Algorithm 2.5. Then it holds that $v(P_F) \geq (1 - 7\varepsilon) \cdot v(\mathrm{OPT}_\mathcal{T})$.*

Proof. Consider P_O, the solution of the ordinary subproblem returned by Algorithm 2.4(including the virtual knapsack and resource augmentation). We know that $v(P_O) \geq (1-\varepsilon) \cdot v(\mathrm{OPT}_O) \geq v(\mathrm{OPT}_{1,O}) - 3\varepsilon \cdot v(\mathrm{OPT}_\mathcal{T})$ by Theorem 2.18 and Lemma 2.21.

Define $\mathrm{OPT}_S \coloneqq \mathrm{OPT}_1 \cap J_S$, and $P_2 \coloneqq P_O \cup \mathrm{OPT}_S \cup J_E$. Then, from the observation that $\mathrm{OPT}_1 = \mathrm{OPT}_{1,O} \cup (\mathrm{OPT}_1 \cap J_E) \cup (\mathrm{OPT}_1 \cap J_S)$, we can deduce

$$\begin{aligned} v(\mathrm{OPT}_1) &= v(\mathrm{OPT}_{1,O}) + v(\mathrm{OPT}_1 \cap J_E) \ + \ v(\mathrm{OPT}_1 \cap J_S) \\ &\leq v(P_O) + 3\varepsilon \cdot v(\mathrm{OPT}_\mathcal{T}) + v(J_E) + v(\mathrm{OPT}_S) \\ &\leq v(P_2) + 3\varepsilon \cdot v(\mathrm{OPT}_\mathcal{T}). \end{aligned}$$

With Observation 2.20, we then obtain $v(P_2) \geq v(\mathrm{OPT}_\mathcal{T}) - 4\varepsilon \cdot v(\mathrm{OPT}_\mathcal{T})$.

We now modify P_2 to obtain a solution P_3 that lacks the virtual ordinary knapsack and deals with bundles instead. Build $\frac{A_S}{\varepsilon}$ equal-sized bundles from P_O as in Step 5. Place these bundles fractionally on the remaining space of the special knapsacks that is left after OPT_S is packed. This space is sufficient by definition of S_O and $\mathcal{P}_1$. Arrange the bundles such that the lowest-value ones are placed fractionally, so removing them from the solution incurs a loss of at most $\varepsilon v(\mathrm{OPT}_\mathcal{T})$. Further, remove the items that were placed fractionally when cutting bundles. Since there are at most $\frac{A_S}{\varepsilon}$ of these and they are at most as valuable as the $\frac{A_S}{\varepsilon^2}$ items in J_E, this incurs a loss of at most $\varepsilon \cdot v(\mathrm{OPT}_\mathcal{T})$.

Therefore, $v(P_3) \geq v(P_2) - 2\varepsilon \cdot v(\mathrm{OPT}_\mathcal{T})$. Moreover, the portion of P_3 on special knapsacks is a valid solution for the request sent to the special subinstance. Therefore, using Theorem 2.11, the overall solution P_F satisfies hte inequality $v(P_F) \geq (1 - 7\varepsilon) \cdot v(\mathrm{OPT}_\mathcal{T})$, as desired. □

Together with Theorem 2.17, this implies that the final solution output by Algorithm 2.5 has a value of at least $(1 - 12\varepsilon) \cdot v(\mathrm{OPT})$.

Lemma 2.23. *Algorithm 2.5 has an update time of at most* $(\frac{\log n}{\varepsilon})^{O(\frac{1}{\varepsilon})} \cdot (\log \overline{v} \cdot \log S_{\max})^{O(1)}$.

Proof. The update time largely results from the combination of update times from the subproblems, i.e., from Theorems 2.11 and 2.18. Note, however, that, since we create the data structures in the special subproblem from scratch with item types, we do not require continuous maintenance of data structures here. Specifically, no data structures for each possible value class are needed, so we lose the dependency on $\overline{\overline{v}}$. In fact, since the number of items in the special subproblem is the sum of the numbers of item types and bundles, which is at most $(\frac{\log n}{\varepsilon})^{O(\frac{1}{\varepsilon})}$, the runtime of the special subproblem is dominated by that of the ordinary one.

We briefly discuss the remaining factors in the running time, and show that they do not increase the time bound. Guessing k contributes a factor of $\frac{1}{\varepsilon}$ to the update time, guessing S_O a factor of $O(\frac{\log(S_{\max} A_S)}{\varepsilon})$, and placing the $\frac{A_S}{\varepsilon^2}$ most valuable ordinary items on extra knapsacks and removing them from data structures takes time $O(\frac{A_S}{\varepsilon^2} \log n)$, all of which is within the time bound. Cutting the items placed in the virtual ordinary knapsack into $\frac{A_S}{\varepsilon}$ equal-sized bundles can be archived efficiently as follows. Compute the total size of these items, using the number of items used for each of the $O(\frac{\log^2 n}{\varepsilon^4})$ item types and deduce the size of a bundle. Sort the item types, e.g., by value then size, and then iteratively pack items of the same type by computing how many items of this type fit in the next non-empty bundle. This takes time $O(\frac{\log^2 n}{\varepsilon^4} \cdot \frac{A_S}{\varepsilon})$, which is sufficient. □

Since queries essentially use the approach from the ordinary the subproblem, the query times are identical to the ones described in Section 2.4.2.

Lemma 2.24. *The query times of Algorithm 2.5 are as follows.*

(i) Single-item queries are answered in time $O(\log(\frac{\log n}{\varepsilon}))$.
(ii) Solution-value queries are answered in time $O(1)$.
(iii) Queries of the entire solution P *are answered in time* $O(\log(\frac{\log n}{\varepsilon})|P|)$.

Proof of Theorem 2.19. To complete the proof of Theorem 2.19, we merely need to combine Lemmas 2.22 to 2.24. □

2.6 Conclusion

We presented efficient dynamic algorithms for the multiple knapsack problem and several special cases that save solutions implicitly and facilitate access to the solution via query operations.

Clearly, it would be interesting to generalize our results beyond multiple knapsack. A natural generalization is d-dimensional knapsack, where the items and knapsacks have a size in each of the d dimensions, and a feasible packing of a subset of items must meet the capacity constraint in each dimension. A reduction to one dimension by [VL81] immediately yields a dynamic $\frac{1-\varepsilon}{d}$-approximation, but designing a dynamic framework with a better guarantee than this remains open. Note that, unless $\mathrm{W}[1] = \mathrm{FPT}$, 2-dimensional knapsack *does not* admit a dynamic algorithm maintaining a $(1-\varepsilon)$-approximation in worst-case update time $f(\varepsilon) \cdot n^{O(1)}$ [KS10].

A recent line of research exploits fast techniques for solving convolution problems to speed up knapsack algorithms (exact and approximate); see, e.g., [Cha18; Jin19; AT19; PRW21; KP04]. In fact, it has been shown that knapsack is computationally equivalent to the $(\min, +)$-convolution problem [CMW+19]. It seems worth exploring whether such techniques are useful in the dynamic setting. Here, it is unclear whether the re-computation of a solution in a new iteration can be done in polylogarithmic time. It is also open whether such techniques can be applied for solving multiple knapsack, even in the static setting.

We hope to foster further research for other packing, scheduling and, generally, non-graph problems in the dynamic setting.

On Packing Anchored Rectangles

3

The *lower-left-anchored rectangle packing (LLARP)* problem is an old mathematical toy problem that dates back to 1969 when it was first posed by Allen Freedman [Tut69] on the 3rd Waterloo Conference on Combinatorics. Given the unit square $U := [0,1]^2$ in the plane and a finite set of points $P \subseteq U$, the task is to find a maximum-area set of axis-aligned and interior-disjoint rectangles $\mathcal{S}$, such that for each $p \in P$, there is exactly one (possibly trivial) rectangle $R \in \mathcal{S}$ that has p in its lower-left corner. Freedman conjectured [Tut69; Win07] that the area covered is at least $\frac{1}{2}$ as long as $(0,0) \in P$. While recent results made progressive improvements, with Damerius et al. [DKK+21] showing that a simple greedy algorithm due to Dumitrescu and Tóth [DT15] covers at least 39% of U, the conjecture still remains open. See Figure 3.1 for an example of a lower-left-anchored rectangle packing and an upper bound matching Freedman's conjecture.

In this chapter, we consider the problem from the lens of approximation algorithms. That is, given an instance P of the LLARP problem, we want to

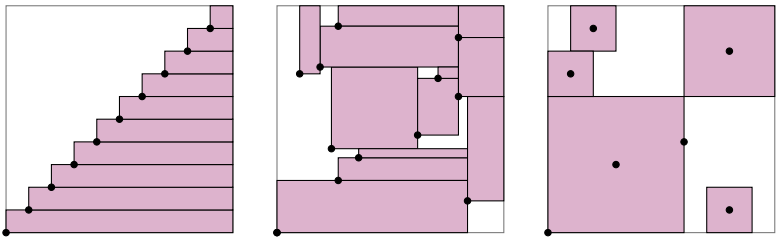


Fig. 3.1: Examples of lower-left-anchored rectangle packings (left and middle), and a center-anchored rectangle packing (right). Instances as the one on the left ($n \to \infty$ equidistant points on the diagonal) serve as an upper bound for the lower-left-anchored rectangle packing matching Freedman's conjecture since, in the limit, an area of at most $\frac{1}{2}$ can be covered.

design a polynomial-time algorithm that finds a solution $\mathcal{S}$ that maximizes the size of the covered area, denoted by $\mathcal{A}(\mathcal{S})$. To measure the performance of the algorithm, we compare $\mathcal{A}(\mathcal{S})$ to OPT_P by which we denote the size of the area covered by an optimal solution to the instance P. If for every instance P, we have $\mathcal{A}(\mathcal{S}) \geq \alpha \cdot \mathrm{OPT}_P$ for some $\alpha \leq 1$, the algorithm is said to be an *α-approximation algorithm*. The largest $\alpha \leq 1$ for which this is true is called the *approximation ratio* of the algorithm. Particularly desirable in this context is the design of a *Polynomial Time Approximation Scheme (PTAS)*, a family of polynomial-time algorithms that compute, for each $\varepsilon > 0$, a $(1-\varepsilon)$-approximate solution.

Even though the formulation of the lower-left-anchored rectangle packing problem and Freedman's conjecture [Tut69] date back to 1969, it is still not even known whether the decision variant of LLARP is NP-complete. Therefore, we consider the complexity of a related packing problem and initiate the investigation of LLARP under use of *resource augmentation*. Resource augmentation describes a setting in which our algorithm is granted some additional capabilities which are not afforded to the optimal solution used to measure the algorithm's performance. We study two different versions of resource augmentation. In the first, the algorithm is allowed to place rectangles with lower-left corners not only exactly at a point in P but also up to an ε distance away from it, which can be thought of as perturbing the anchor. In the second version, the rectangle set produced by the algorithm may have some (bounded) overlap. In both cases, the resulting solution is compared to an optimal solution without these capabilities. For the first setting, we develop an algorithm that produces an anchored rectangle set of total area no less than that of an optimal solution of LLARP without resource augmentation. Our analysis is combinatorial in nature and consists of transforming the optimal solution to a feasible solution for a perturbed instance by using a specific linear program with totally unimodular incidence matrix. For the second setting, we provide an algorithm that covers at least $(1-\varepsilon)$ times the area of an optimal solution. It builds on the algorithm for the first version of resource augmentation. These results are given in Section 3.2.

We also introduce a natural generalization of the LLARP problem that looks at anchors different from the lower-left corner. Specifically, we use *anchorings* defined by a pair $(\alpha, \beta) \in [0,1]^2$: The *$(\alpha, \beta)$-anchored rectangle packing*

(ARP) problem looks for a maximum-area rectangle set as above but with rectangles anchored in the relative position (α, β), with $(0, 0)$-ARP being equal to the LLARP problem, while $(\frac{1}{2}, \frac{1}{2})$-ARP requires all rectangles to have a $p \in P$ at their center. We refer to the latter as the *center-anchored rectangle packing (CARP)* problem, an example of which can be found in Figure 3.1. We give a detailed definition of the (α, β)-ARP problem in Section 3.1.

When looking at the complexity of CARP, a difference that stands out in comparison with LLARP is that CARP allows one to simulate "non-expandable" points: By putting four points at the corners of a tiny ε-sized square, none of their rectangles' side lengths can exceed ε. One can think of these four points as one input point that cannot be used as an anchor but still restricts the expansion of other rectangles. These non-expandable points turn out to be a valuable asset, as they can be used to build walls and inject additional geometry into the problem. This can be exploited to encode maximum independent set into a CARP instance, proving NP-hardness of CARP. The construction of non-expandable points seems difficult or even impossible for LLARP and is the main obstacle in transferring the NP-hardness proof to that setting. For the NP-hardness proof, we refer the reader to [ABC+19] and the PhD thesis of Christoph Damerius.

Complementing the NP-hardness of CARP, we develop a PTAS for CARP in Section 3.3, which also extends to any anchoring $(\alpha, \beta) \in (0, 1)^2$. The PTAS is based on a carefully constructed input instance for a related problem called *maximum weight independent set of rectangles* (MWISR) and the usage of a known PTAS in a resource augmentation setting of MWISR.

Related Work. After being seemingly forgotten for decades, the lower-left-anchored rectangle packing problem surfaced again in a puzzle of the IBM "ponder this" challenge [IBM04] and in a book comprising several mathematical puzzles by Peter Winkler [Win07]. Since then, there has been a resurgence of interest in the problem, as well as related variants. The best known polynomial time algorithm for the LLARP problem is a simple greedy algorithm due to Dumitrescu and Tóth [DT15] that orders the points $(x, y) \in P$ decreasingly according to the key value $x + y$ and, in this order, chooses legal rectangles of maximum size anchored in the respective point. They show that this algorithm covers an area of at least 0.091. Since the optimal solution

cannot cover more than an area of 1 (the whole unit square), their analysis also implies a 0.091-approximation. Damerius et al. [DKK+21] improve the analysis of this algorithm and show that in fact at least 39% of U is covered. They also show that this greedy algorithm can not be the answer to the conjecture by giving an instance in which it covers less than 43.3%. Another interesting question is how large an area can one expect to cover when the input points are chosen uniformly at random. Maat [HM21] and Haupt [Hau21] consider this question, and Haupt gives an algorithm that, in expectation, covers an area of 0.6.

In the setting where we must pack squares instead of arbitrary rectangles, Balas et al. [BDT17] achieve an approximation ratio of $\frac{1}{3}$. They also consider a setting where the algorithm can choose for each rectangle from multiple anchors, its four corners, and give a $(\frac{7}{12} - \varepsilon)$-approximation algorithm with $\varepsilon = |P|^{-1}$ and a Quasi-Polynomial Time Approximation Scheme (QPTAS) for rectangles, as well as a $\frac{9}{47}$-approximation algorithm and a PTAS for squares. The QPTAS and PTAS extend to the lower-left anchored variants. Recently, Akitaya et al. [AJS+18] gave the first NP-hardness result for an ARP variant where only squares may be packed and each square can be anchored at any of its four corners. They also show that, for any instance consisting of finitely many input points inside U, the union of all feasible anchored square packings covers an area of at least $\frac{1}{2}$. Finally, if rectangles can be anchored at any of their four corners but input points are restricted to the boundary of U, Biedl et al. [BBM+17] give an polynomial-time algorithm (based on maximum independent set for a specific class of graphs) that finds an optimal solution.

ARP problems fall within the more general setting of packing axis-aligned and interior-disjoint rectangles in a rectangular container. This setting captures several important and well studied optimization problems. See, for example, the (NP-hard) problems 2D-knapsack and strip packing [AW13; BK14], as well as maximum area independent set of rectangles [AW15; BDJ10a; BDJ10b]. Similar problems have also been formulated by Radó and Rado [Rad28; Rad49; Rad51; Rad68]. These problems differ from ARP in that the size of the packed objects is part of the input and not controllable and, in some cases, there is no anchoring of the rectangles.

3.1 Preliminaries

We start with some general notation and the formal definitions of the problems studied in this chapter. We use $U = [0,1]^2$ to denote the unit square. Consider a point $p \in U$, an axis-aligned rectangle $R \subseteq U$, and its lower-left corner $(x, y) \in R$. Let w and h denote the width and height of R, respectively. For a pair $(\alpha, \beta) \in U$, called *anchoring*, we say R is (α, β)-*anchored* in p if $p = (x + \alpha \cdot w, y + \beta \cdot h)$. In this case, we also say a point (α, β)-*spans* R and call it the *anchor* of the rectangle. If the anchoring (α, β) is clear from context, we may omit it. We use $\mathcal{A}(R) = w \cdot h$ to denote the area of R. Similarly, given a set $\mathcal{S}$ of rectangles, we define $\mathcal{A}(\mathcal{S}) = \sum_{R \in \mathcal{S}} \mathcal{A}(R)$. If not stated otherwise, any rectangle is assumed to be axis-aligned.

The Anchored Rectangle Packing Problem. We now define the *anchored rectangle packing (ARP)* problem with respect to an anchoring $(\alpha, \beta) \in U$. An instance of the ARP problem consists of a finite set $P \subseteq U$ of n points. A *valid rectangle set* (also *solution*) $\mathcal{S}$ for the instance P is a set of n interior-disjoint rectangles that, for each $p \in P$, contains one possibly zero-size rectangle R_p such that R_p is (α, β)-anchored in p and does not contain any point from $P \setminus \{p\}$ in its interior. The goal is to find a solution $\mathcal{S}$ for P that covers as much area as possible. We use $\mathcal{S}^*_{\text{ARP}(P)}$ to denote a valid rectangle set of maximum area $\text{OPT}_{\text{ARP}(P)} := \mathcal{A}(\mathcal{S}^*_{\text{ARP}(P)})$ and omit ARP and/or P if it is clear from context.

We mostly consider the two following anchorings: The anchoring $(0, 0)$, for which we refer to the problem by the name *lower-left-anchored rectangle packing (LLARP)*, and the anchoring $(\frac{1}{2}, \frac{1}{2})$, for which we refer to the problem by the name *center-anchored rectangle packing (CARP)*.

3.2 Resource Augmentation

In this section, we investigate the lower-left-anchored rectangle packing problem with two types of resource augmentation. We start by defining the two types of resource augmentation and stating the respective results.

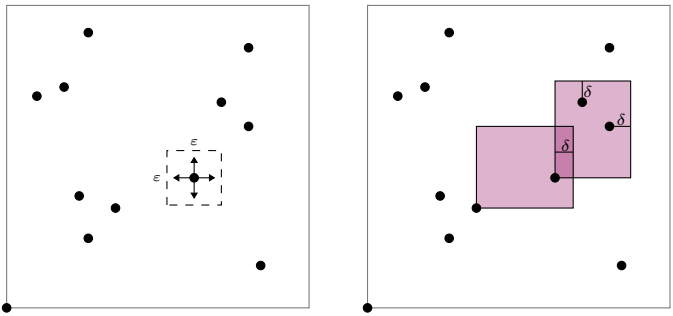


Fig. 3.2: Illustrations of the two resource augmentation types perturbation (left) and overlap (right).

Perturbation Augmentation. In this type of resource augmentation, a solution rectangle does not need to be anchored exactly in the corresponding point. Instead, its anchor may be up to an ε distance away from that point (see Figure 3.2, left), for some $\varepsilon > 0$. Formally, denote by $\mathrm{dist}_\infty(x, y)$ the distance of two points $x, y \in \mathbb{R}^2$ in the ℓ_∞-norm. For $X \subseteq \mathbb{R}^2$ and $y \in \mathbb{R}^2$, we write $\mathrm{dist}_\infty(y, X) = \inf_{x \in X} \mathrm{dist}_\infty(y, x)$ and denote by $\overline{X} = \mathbb{R}^2 \setminus X$ the set's complement. Denoting by $\ell(R_p)$ the lower-left corner of a rectangle R_p, we say that a set of interior-disjoint rectangles $\{R_p\}_{p \in P}$ is *ε-valid*, if $\mathrm{dist}_\infty\big(p, \ell(R_p)\big) < \varepsilon$ and $\mathrm{dist}_\infty\big(p, \overline{R_p}\big) < \varepsilon$ for all $p \in P$. Thus, in the perturbation-augmentation, an algorithm may output an ε-valid solution while an optimal solution is required to be a valid rectangle set as described in Section 3.1.

We now define an *ε-grid* Γ. This is a pair $(V, \mathcal{L})$ that consists of a set of *grid points* $V = \{(x, y) \in U \mid x = \varepsilon \cdot k_x \ \wedge \ y = \varepsilon \cdot k_y \ \wedge \ k_x, k_y \in \mathbb{N}\}$, together with a set of *grid cells* $\mathcal{L} = \big\{[x, x + \varepsilon] \times [y, y + \varepsilon] \subseteq U \mid (x, y) \in V\big\}$. The perturbation-augmentation allows us to focus on solutions where all vertices of rectangles are points on an ε-grid Γ, thereby allowing us to enumerate all possible sets of interior-disjoint, axis-aligned rectangles with vertices in V. We call such an solution a *grid-point solution*. We show that (i) there exists a polynomial-time algorithm that finds an optimal ε-valid grid-point solution, and (ii) this solution covers at least as much area as an optimal valid rectangle set without resource augmentation. This implies the following theorem.

Algorithm 3.1 Algorithm using resource augmentation of perturbation-type

1: $\Gamma \leftarrow \varepsilon$-grid in U, $\mathcal{H} \leftarrow \emptyset$
2: **for** every ε-valid configuration C for P on Γ **do**
3: **for** every grid-point solution $\mathcal{S} := \{R_q\}_{q \in C}$ of LLARP(C) **do**
4: $\mathcal{H} \leftarrow \mathcal{H} \cup \{\mathcal{S}\}$
5: **return** $\mathcal{S} \in \mathcal{H}$ maximizing $\mathcal{A}(\mathcal{S})$

Theorem 3.1. *For every $\varepsilon > 0$ and given a finite point-set $P \subseteq U$, there exists a polynomial time algorithm that computes an ε-valid set of interior-disjoint rectangles that covers an area of at least $\mathrm{OPT}_{\mathit{LLARP}(P)}$.*

Overlap Augmentation. This type of resource augmentation relaxes the condition that the rectangles need to be disjoint. Specifically, each pair of rectangles is allowed to overlap by a thin strip of width at most δ. Note that this implies that a rectangle may also contain multiple input points as long as they are no more than a δ-distance away from its boundary (see Figure 3.2, right). Formally, we shall call a set of rectangles $\{R_p\}_{p \in P}$ a *δ-LLARP*, if the rectangle R_p is lower-left-anchored in p and $\sup_{x \in R_{p'}} \mathrm{dist}_\infty\big(x, \overline{R}_p\big) < \delta$, for all $p' \in P \setminus \{p\}$. We show that we can transform an ε-valid grid-point solution into a δ-LLARP while only losing a small fraction of the covered area. Naturally, any area that is covered by multiple rectangles is counted only once.

Theorem 3.2. *For any $\varepsilon > 0$, there exists a polynomial time algorithm, that, given a finite point-set $P \subseteq U$, outputs an $\frac{\varepsilon}{11}$-LLARP covering an area of at least $(1-\varepsilon) \cdot \mathrm{OPT}_{\mathit{LLARP}(P)}$.*

We proceed to prove these results, starting with Theorem 3.1.

Proof of Theorem 3.1. Let $\Gamma = (V, \mathcal{L})$ be an ε-grid in U and denote by k the number of grid cells. Without loss of generality, assume that all points of P lie in the interior of one of the $O(\varepsilon^{-2})$ grid cells. A *grid configuration* on Γ is a subset of the grid points $C \subseteq V$. We say that a grid configuration C on Γ is *ε-valid* for P if there is a surjective mapping $\varphi : P \to C$, such that $\mathrm{dist}_\infty\big(p, \varphi(p)\big) \leq \varepsilon$. In other words, ε-valid grid configurations are precisely those grid-point sets that can be obtained from P via perturbation

resource augmentation. Note that any LLARP for an ε-valid configuration C for P can be extended to an ε-valid solution for P by adding degenerate rectangles anchored at points in C.

Our algorithm, summarized as Algorithm 3.1, enumerates all grid configurations on Γ that are ε-valid for P. Then, for each such configuration C, it computes all grid-point solutions $\{R_q\}_{q \in C}$ of LLARP(C). Among all enumerated solutions, the one that maximizes the covered area is kept as the final solution.

First, we show that the running time of this algorithm is polynomial. Note that any grid cell has four grid points as its vertices. Since any point $p \in P$ lies in the interior of some grid cell $L \in \mathcal{L}$, an ε-valid grid configuration for P must contain at least one of the four vertices of L. Thus, we can construct any ε-valid grid configuration by choosing for each grid cell $L \in \mathcal{L}$ and each input point $p \in P \cap L$, a vertex of L as its image in the mapping φ. However, whenever multiple rectangles are anchored in the same point at most one can be non-degenerate. Thus, in each $L \in \mathcal{L}$, it suffices to decide on one of at most 15 cell configurations using either 0, 1, 2, 3, or 4 of the grid cell's vertices, where 0 vertices are used if and only if $L \cap P = \emptyset$. In particular, any number of contained input points larger than 3, yields the same set of $15 = 4^4 - 1$ possible cell configurations. Thus, by counting the number of input points contained in each grid cell (in time $O(n)$) and then enumerating at most 15^k grid configurations, we obtain all ε-valid grid configurations. For each such configuration C, we obtain all corresponding ε-valid grid-point solutions by picking for each $q \in C$ one of the $O(\varepsilon^{-2})$ grid points above and to the right of q as the upper-right corner of the rectangle and checking interior disjointness. This amounts to at most $O(\varepsilon^{-2\varepsilon^{-2}})$ solutions per grid configuration. We conclude that the running time of Algorithm 3.1 is linear in n and doubly exponential in $\frac{1}{\varepsilon}$.

It remains to show that the computed grid-point solution covers at least as much area as an optimal lower-left-anchored rectangle packing without resource augmentation $\mathcal{S}^*_{\text{LLARP}(P)} = \{R_1, R_2, \ldots, R_n\}$. To this end, we transform $\mathcal{S}^*_{\text{LLARP}(P)}$ to a grid-point solution by "snapping" the corners of its rectangles to Γ in a manner that at least preserves the covered area. We do this by first snapping the lower-left and upper-right corners of each rectangle to either of the horizontal grid-lines directly above and below the corner in

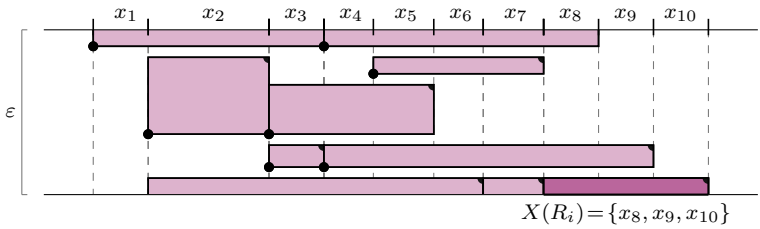


Fig. 3.3: A row B of the ε-grid Γ and the corresponding rectangles of $\mathcal{S}^*_B$.

question. Afterwards, the same is done analogously for the vertical grid-lines, which we omit in our discussion.

Consider a row of the grid, say $B = [0,1] \times (y, y+\varepsilon)$, with $(x,y) \in V$ for some x, see Figure 3.3 for an illustration. We describe how the lower-left and upper-right corners within this row can be snapped up or down without reducing the total area. Partition the row horizontally into segments $x_1, x_2, \ldots, x_r$ in-between the x-coordinates of corners of those rectangles of $\mathcal{S}^*_B := \{R_i\}_{i=1}^m \subseteq \mathcal{S}^*_{\text{LLARP}(P)}$ which have a corner in the row. We associate a variable $h_i \in [0,1]$ with each rectangle R_i denoting the height of $R_i \cap B$ relative to the height of B. Snapping the rectangles in this row can be seen as rounding each such variable h_i either up to 1 or down to 0 (i.e., expanding the rectangle to span the full row height or collapsing it such that it does not appear in the row anymore). We aim to do this in a manner, which neither introduces overlaps nor decreases the covered area. This snapping problem is represented by the following integer linear program.

$$\max \quad \sum_{i=1,\ldots,m} h_i \cdot w(R_i) \tag{3.1}$$

$$\text{s.t.} \quad \sum_{i\,:\,x_j \in X(R_i)} h_i \leq 1 \qquad \text{for } j = 1, \ldots, r, \tag{3.2}$$

$$h_i \in \{0,1\} \qquad \text{for } i = 1, \ldots, m. \tag{3.3}$$

Here, $w(R_i)$ denotes the width of rectangle R_i, and the set $X(R_i)$ consists of all segments x_i intersecting R_i. The objective function (3.1) maximizes the covered area while constraints in (3.2) ensure that, in an integer solution, rectangles do not overlap. The binary constraints in (3.3) force the rectangles

Algorithm 3.2 Algorithm using resource augmentation of the overlap-type

1: Compute $S_1 \leftarrow$ Algorithm 3.1$(P, \frac{\varepsilon}{22})$
2: **for** all $R_p \in S_1$ **do**
3: $\quad R'_p \leftarrow$ the rectangle spanned by p and $u(R_p) + (\frac{\varepsilon}{22}, \frac{\varepsilon}{22})$
4: $\quad$ **if** $u(R'_p) \notin U$ **then**
5: $\quad\quad u(R'_p) \leftarrow \arg\min_{x \in U} ||x - u(R'_p)||$
6: **return** $S_{\text{II}} = \{R'_p\}_{p \in P}$

to either span the entire height of the row or none of it. We note that the linear program is inspired by a similar one which is used for the demand flow problem [CMS07].

It is easy to verify that $\mathcal{S}_B^*$ corresponds to a feasible solution for the linear programming relaxation of (3.1)–(3.3) and thus lower bounds the optimal value of the LP-relaxation. Since the sets $X(R_i)$ contain consecutive line segments, the LP satisfies the consecutive ones property [Sch99]. Therefore, constraints (3.2) yield a totally unimodular matrix and the LP is integral [Sch99]. It follows, that there is an integral solution that solves the relaxed linear program optimally and induces a partial snapping of area at least $\text{OPT}_{\text{LLARP}(B)}$. Repeating this argument for all rows and columns of Γ yields a grid-point solution with an area of at least $\text{OPT}_{\text{LLARP}(P)}$. □

We now proceed to prove Theorem 3.2 for the resource augmentation of the overlap type.

Proof of Theorem 3.2. We describe an algorithm, which is summarized as Algorithm 3.2, that satisfies the conditions of Theorem 3.2. That is, the algorithm runs in polynomial time and outputs an $\frac{\varepsilon}{11}$-LLARP that covers an area of at least $(1 - \varepsilon) \cdot \text{OPT}_{\text{LLARP}(P)}$. Recall that, as opposed to the perturbation-augmentation, input points cannot be moved, but the rectangles may somewhat overlap.

Consider the $\frac{\varepsilon}{22}$-valid solution $\mathcal{S}_1 = \{R_p\}_{p \in P}$ of Algorithm 3.1 obtained by using as input the instance P and the approximation parameter $\varepsilon_1 = \frac{\varepsilon}{22}$. When constructing $\mathcal{S}_1$ via Algorithm 3.1, we move each input-point $p \in P$ to some grid point q_p of an ε_1-grid to span the rectangle R_p, the upper-right corner of which we denote by $u(R_p)$. We transform the solution of Algorithm 3.1 to obtain an $\frac{\varepsilon}{11}$-LLARP.

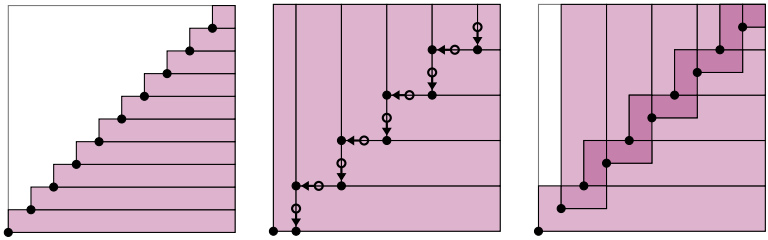


Fig. 3.4: Solutions of Algorithms 3.1 and 3.2 with $\varepsilon = \frac{1}{n}$ for a simple instance illustrate the potential of the two kinds of resource augmentation.

Shift each $u(R_p)$ up and to the right by $\frac{\varepsilon}{22}$ to obtain $u(R_p)'$. The transformed solution $\mathcal{S}_1'$ consists of the rectangles defined by the lower-left corners $p \in P$ and the corresponding upper-right corners $u(R_p)'$. Since $\mathrm{dist}_\infty(p, q_p) < \frac{\varepsilon}{22}$, moving $u(R_p)$ by $(\frac{\varepsilon}{22}, \frac{\varepsilon}{22})$ ensured that transforming the solution did not decrease its size. The overlap of rectangles is bounded by $2 \cdot \frac{\varepsilon}{22} = \frac{\varepsilon}{11}$.

However, moving the solution may cause some rectangles to protrude from the unit square. We prune this excess area, by moving the respective upper-right corners back into the square to obtain our final solution $\mathcal{S}_{\mathrm{II}}$. Due to this pruning, we lose an area of size at most $2 \cdot \frac{\varepsilon}{22} = \frac{\varepsilon}{11} \leq \varepsilon \cdot \mathrm{OPT}_{\mathrm{LLARP}(P)}$. The inequality follows from the lower bound of $\frac{1}{11}$ on $\mathrm{OPT}_{\mathrm{LLARP}(P)}$ [DT15]. This implies, that $\mathcal{A}(\mathcal{S}_{\mathrm{II}}) \geq (1 - \varepsilon) \cdot \mathrm{OPT}_{\mathrm{LLARP}(P)}$, concluding the proof of Theorem 3.1. □

3.3 A PTAS for ARP with Fractional Anchorings

In this section, we give a PTAS for the anchored rectangle packing problem for any anchoring $(\alpha, \beta) \in (0, 1)^2$. It is based on a result for *maximum weight independent set of rectangles (MWISR)* that uses resource augmentation similar to that in Theorem 3.2. We note that a similar connection to MWISR has recently been used to obtain a PTAS for packing squares and a QPTAS for packing rectangles when the anchors are allowed to lie in any of the four corners (and may be different for each rectangle) [BDT17]. An instance of MWISR consists of a set of n axis-aligned, weighted rectangles.

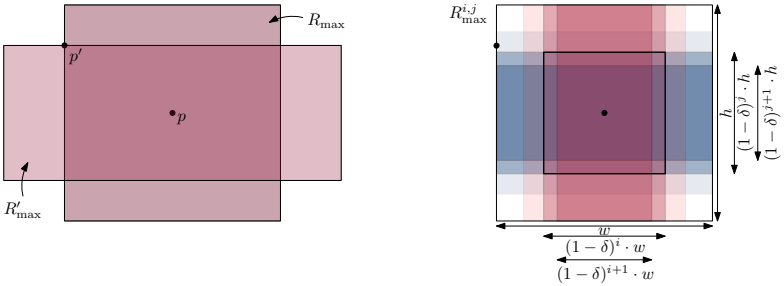


Fig. 3.5: Maximal rectangles $R_{\max}$ and $R'_{\max}$ (left) and candidate rectangles $R^{i,j}_{\max}$ derived from $R_{\max}$ for the anchoring $(\frac{1}{2}, \frac{1}{2})$ (right).

The goal is to compute a maximum-weight subset of pairwise interior-disjoint rectangles. In *δ-MWISR*, a variant of MWISR using resource augmentation, rectangles must be non-overlapping only *after* shrinking them by a factor of $1 - \delta$, where $\delta > 0$ is a fixed parameter. Here, Adamaszek et al. [ACW15] give an algorithm that, in time $n^{(\delta\varepsilon)^{-O(1/\varepsilon)}}$, computes a solution which is within a $(1 - \varepsilon)$-factor of the optimum MWISR solution. While the original algorithm considers shrinking only around the rectangles' centers, we can make some adjustments and then use it to obtain a PTAS for arbitrary anchorings $(\alpha, \beta) \in (0, 1)^2$. However, this approach cannot be adapted for anchorings on the boundary of U, in particular, not for LLARP as we see in the proof of the following theorem.

Theorem 3.3. *For any fixed anchoring $(\alpha, \beta) \in (0, 1)^2$, the anchored rectangle packing problem admits a polynomial-time approximation scheme.*

Proof. Consider an instance $P \subseteq U$ of (α, β)-ARP with n input-points and fix some $\varepsilon > 0$. Assume, without loss of generality, that $\alpha \leq \beta \leq \frac{1}{2}$, and define $N := \lceil \log_{1-\varepsilon}(\varepsilon/n) \rceil + 1$ and $\delta := 2\alpha\varepsilon$.

Observe that, for each pair $p, p' \in P$ of input points, there are at most two inclusion-wise maximal rectangles that are anchored in p and have p' on their boundary, one where p' is on the left or right side of the rectangle and one where p' is on its bottom or top side, see Figure 3.5 (left). Thus, there are at most $2n$ inclusion-wise maximal rectangles anchored in a point $p \in P$ and at most $2n^2$ such rectangles overall. We call them *maximal rectangles*.

Algorithm 3.3 PTAS for (α, β)-ARP with $\varepsilon < 0$

1: $N \leftarrow \lceil \log_{1-\varepsilon}(\varepsilon/n) \rceil + 1$, $\delta \leftarrow 2\alpha\varepsilon$
2: $\mathcal{C} \leftarrow \{R^{i,j}_{\max} \mid R_{\max}$ maximal rectangle, $i, j = 0, \ldots, N-1\}$
3: $\mathcal{S} \leftarrow$ solution to δ-MWISR instance $(\mathcal{C}, \mathcal{A})$ using the algorithm from [ACW15]
4: **return** $\mathcal{S}' = \{R^{i+1,j+1}_{\max} \mid R^{i,j}_{\max} \in \mathcal{S}\}$

In order to obtain a PTAS, construct for each maximal rectangle $R_{\max}$ additional N^2 *candidate rectangles* $(R^{i,j}_{\max})_{i,j=0}^{N-1}$ by scaling $R_{\max}$, with respect to the anchoring (α, β) by a factor of $(1-\varepsilon)^i$ along the horizontal and $(1-\varepsilon)^j$ along the vertical axis, respectively. See Figure 3.5 (right) for an illustration of this. Denote the set of all such candidate rectangles by $\mathcal{C}$.

Apply the PTAS for δ-MWISR from [ACW15], with approximation parameter ε and shrinking parameter δ, to the MWISR instance $(\mathcal{C}, \mathcal{A})$ consisting of all candidate rectangles weighted by their area. We obtain a solution $\mathcal{S}$ for δ-MWISR that consists of one rectangle R_p for each anchor p (after possibly adding some empty rectangles). By definition of δ-MWISR, shrinking these rectangles by a factor of $1-\delta$ around their center yields a set of independent rectangles. Due to this shrinking around the center, however, they may not be (α, β)-anchored in their corresponding anchor anymore. Our algorithm outputs, for each $R_p = R^{i,j}_{\max} \in \mathcal{S}$, the rectangle $R^{i+1,j+1}_{\max}$, or an empty rectangle if R_p is empty. For a summary of the algorithm, see Algorithm 3.3.

To show that this output is indeed a valid solution for our (α, β)-ARP instance, first note that the rectangles are (α, β)-anchored in input points, since they are candidate rectangles. Further, we show that, for appropriately chosen δ, a rectangle $R^{i+1,j+1}_{\max}$ is contained in the rectangle $R^{i,j}_{\max}$ even after the latter was shrunk by a factor of $1-\delta$ around its center. With our assumption in mind that $\alpha \leq \beta \leq \frac{1}{2}$, it is easy to see that this holds if and only if $\alpha\varepsilon \geq \frac{1}{2}\delta$, which is equivalent to $\delta \leq 2\alpha\varepsilon$. Therefore, by definition of δ-MWISR, the output rectangles form an independent set and thus a valid (α, β)-ARP solution. Observe that the requirement $\delta \leq 2\alpha\varepsilon$ clearly shows that this approach cannot work for anchorings on the boundary of U, as in the lower-left-anchored rectangle packing problem.

It remains to analyze the approximation ratio of Algorithm 3.3. The rectangle set $\mathcal{S}$ that is obtained by the algorithm for δ-MWISR covers an area

of size $\mathcal{A}(\mathcal{S}) \geq (1-\varepsilon) \cdot \mathrm{OPT}_{\mathrm{MWISR}}$, where $\mathrm{OPT}_{\mathrm{MWISR}}$ denotes the area covered by an optimal solution for the instance $(\mathcal{C}, \mathcal{A})$ of MWISR. Scaling each rectangle in $\mathcal{S}$ by $1-\varepsilon$ with respect to the anchoring (α, β), we obtain the rectangle set $\mathcal{S}'$ which is the output of our algorithm. The area covered by $\mathcal{S}'$ is then lower bounded by

$$\mathcal{A}(\mathcal{S}') \geq (1-\varepsilon)^3 \cdot \mathrm{OPT}_{\mathrm{MWISR}}. \tag{3.4}$$

We now bound the area $\mathrm{OPT}_{\mathrm{ARP}}$ of an optimal (α, β)-ARP solution $\mathcal{S}^*_{\mathrm{ARP}}$ in terms of $\mathrm{OPT}_{\mathrm{MWISR}}$. To this end, we fix a rectangle $R^*_p \in \mathcal{S}^*_{\mathrm{ARP}}$ anchored in $p \in P$. This rectangle is contained in at least one maximal candidate rectangle $R_{\max} \in \mathcal{C}$. Let $i, j \in \{0, 1, \ldots, N-1\}$ be maximal such that $R^*_p \subseteq R^{i,j}_{\max}$. Note that we have $\mathcal{A}(R^*_p) \leq \mathcal{A}(R_{\max}) \cdot (1-\varepsilon)^{i+j}$. We say that R^*_p is *negligible* if $i = N-1$ or $j = N-1$. For each non-negligible $R^*_p \in \mathcal{S}^*_{\mathrm{ARP}}$, define $R_p \coloneqq R^{i+1,j+1}_{\max}$ and denote by $\mathcal{S}''$ the set of all such rectangles. We consider the contribution of non-negligible and negligible rectangles to $\mathrm{OPT}_{\mathrm{ARP}}$ separately.

By construction, the non-negligible rectangles cover a total area of at most $(1-\varepsilon)^{-2} \cdot \mathcal{A}(\mathcal{S}'')$. Furthermore, since $\mathcal{S}'' \subseteq \mathcal{C}$ and as the rectangles in $\mathcal{S}''$ are pairwise non-overlapping (as a shrunken subset of an ARP solution), $\mathcal{S}''$ is a solution to the MWISR instance $\mathcal{C}$. This implies that $\mathcal{A}(\mathcal{S}'') \leq \mathrm{OPT}_{\mathrm{MWISR}}$, which bounds the contribution of non-negligible rectangles to $\mathrm{OPT}_{\mathrm{ARP}}$ by $(1-\varepsilon)^{-2} \cdot \mathrm{OPT}_{\mathrm{MWISR}}$.

To bound the contribution of negligible rectangles, fix a negligible $R^*_p \in \mathcal{S}^*_{\mathrm{ARP}}$ and a maximal rectangle $R_{\max}$ containing R^*_p. Note that

$$\mathcal{A}(R^*_p) \leq \mathcal{A}(R_{\max}) \cdot (1-\varepsilon)^{N-1} \leq \mathrm{OPT}_{\mathrm{MWISR}} \cdot (1-\varepsilon)^{N-1} \leq \mathrm{OPT}_{\mathrm{MWISR}} \cdot \varepsilon / n,$$

where the penultimate inequality holds since $\{R_{\max}\}$ is a valid MWISR. As there are at most n negligible rectangles in $\mathcal{S}^*_{\mathrm{ARP}}$, they contribute at most $\varepsilon \cdot \mathrm{OPT}_{\mathrm{MWISR}}$.

Combining the contribution of negligible and non-negligible rectangles, we get $\mathrm{OPT}_{\mathrm{ARP}} \leq (\varepsilon + (1-\varepsilon)^{-2}) \cdot \mathrm{OPT}_{\mathrm{MWISR}}$. With Equation (3.4), this implies

$$\mathcal{A}(\mathcal{S}) \geq (1-\varepsilon)^3 \cdot (\varepsilon + (1-\varepsilon)^{-2})^{-1} \cdot \mathrm{OPT}_{\mathrm{ARP}} \geq (1-\varepsilon)^6 \cdot \mathrm{OPT}_{\mathrm{ARP}},$$

where the last inequality holds for $\varepsilon < 1$.

Concerning the running time, note that for fixed ε, the total time spent to obtain $\mathcal{S}$ is polynomial in n. The bottleneck is the computation of $\mathcal{S}'$ from the $|\mathcal{C}| \leq 2n^2 \cdot N^2$ candidate rectangles. By [ACW15, Theorem 1], this can be done in time $n^{(\delta\varepsilon)^{-O(1/\varepsilon)}}$. □

3.4 Conclusion

In this chapter, we gave two algorithms for the lower-left-anchored rectangle packing problem that use different kinds of resource augmentation to solve the problem optimally and near-optimally, respectively. To our knowledge, this is the first result for this problem that uses resource augmentation. For the still open question regarding the hardness of LLARP, our contribution is the design of more general, related problem, the anchored rectangle packing problem. Here, we give a PTAS for all anchors in $(0, 1)^2$, that is complemented by the NP-hardness proof for the CARP problem in [ABC+19]. As a final observation, we note that when considering the perturbation-augmentation setting for LLARP with resource augmentation, we may obtain a $(1 - \varepsilon)$-approximation using Algorithm 3.3. This is done by considering for each maximal rectangle not its lower-left corner as an anchor but instead a point slightly closer to the center of the rectangle, i.e., the anchor $(\varepsilon, \varepsilon)$, and then scaling the candidate rectangles accordingly and proceeding with the algorithm as usual. The drawback is that, unlike with Theorem 3.1, we only obtain a $(1 - \varepsilon)$ approximation. Also, the running time may depend on the input points, as the existence of a small maximal rectangle upper bounds the shrinking parameter δ which influences the running time. It remains an intriguing problem to design a good approximation algorithm or lower bounds for the LLARP problem and of course to determine whether indeed half of the unit square can always be covered.

4 Simultaneous Allocation and Maintenance Scheduling of Recyclable Resources under Uncertainty with Application in Steelmaking

Timely allocation and maintenance of scarce resources is a challenging but essential task that arises in many production environments. In this chapter, we consider an application in steelmaking, an industrial sector, where rising economic pressure, due to fierce global competition, increasing energy prices, and climate-change-related regulation, has recently led companies to embrace digitalization and concurrent optimization of production processes to boost sustainability and profitability [BFC+20a]. Specifically, we simultaneously consider the maintenance scheduling of working rolls in a hot rolling mill, the allocation of working rolls to production jobs, and the scheduling of these production jobs on a single production processor. We optimize a bi-criteria objective in order to minimize both, the idle time of the production processor and the cost associated with the working-roll-to-production-job allocation. Moreover, we investigate how to deal with the uncertainty that arises from knowing only the next k production jobs in advance.

Given the undeniable importance of maintaining machines and other resources in industrial production, there is a large and still growing body of research devoted to maintenance strategies, with the first operation research models for optimizing maintenance dating back to the sixties [Dek95; BP96; BH60]. Generally, these are difficult problems, especially when combining production scheduling and maintenance. Even the comparatively simple setting of scheduling jobs and preventative machine maintenance tasks on a single machine is strongly $\mathcal{NP}$-hard [QCT99]. However, integrating maintenance planning with production scheduling increases the overall effectiveness of production [NML+03], while failure to consider maintenance may lead to

serious and costly interruptions in the production process [CCC+06]. The recent survey by Geurtsen et al. [GDA+22] considers the topic of integrating production scheduling with resource constraints, and maintenance considerations. It concludes that the combinations of production scheduling with maintenance considerations and with resource constraints, respectively, are widely investigated, while research on the combination of all three is comparatively scarce. However, there is ample evidence that, by combining those different aspects of production in the decision-making process, improvements can be made when compared to production scheduling without taking into account maintenance and/or resources. Thus, investigating the combination of production, resources, and maintenance is an important task that is interesting for both theoreticians and practitioners.

Most scientific publications that study the integration of maintenance and production scheduling only consider the maintenance of machines but not that of resources. A notable exception is the study of injection mould maintenance in plastic production. Here, often all three aspects, production, resources and maintenance, are considered simultaneously, as well as the maintenance of both, machines and resources (i.e., moulds), see, e.g., [WCC12; WCC14; FCN+19]. We are only aware of one work by Wang and Ming [WL15] that also considers the allocation of resources to production jobs. There, however, the allocation influences the job processing times and does not, as in our case, lead to an additional optimization objective. Another difference is found in the type of maintenance that is considered. While the aforementioned articles mostly consider preventative maintenance, i.e., deciding on when to maintain resources in order to prevent complete breakdowns while maximizing the availability of the resource, in our setting, a resource needs to be maintained after each use in a production job.

When considering the combination of production scheduling and resource constraints, the Resource Constrained Project Scheduling Problem (RCPSP) is a prominent example. It is a classical NP-hard [BLK83] optimization problem in which a set of jobs with precedent constraints needs to be scheduled subject to resource constraints in order to minimize an objective, e.g., the makespan. As a long-established challenging problem and with its wide practical applicability it has gathered a large research interest. The classical types of resources considered in RCPSP are renewable, non-renewable or doubly

constrained resources. Over the years, different additional resource types have been proposed, some of which come close to our setting. Somewhat related are changeover resources [NSZ06], allocatable resources [ST03], auxiliary resources [MWW06], complementary resources [AR01], and delay renewable resources [AB97]. The closest model to ours is that of recyclable resources by Shewchuk and Chang [SC95], explaining our choice in naming. Here, too, resources need to be actively recycled on dedicated machines, but the setting does not reach the complexity of our maintenance model. Generally, the focus in RCPSP mostly lies in scheduling the production jobs while we are more focused on resource maintenance and allocation. For a broad introduction to RCPSP, see, e.g., the book [ADN13].

Classical research in production scheduling – and in optimization in general – assumes that an algorithm has full knowledge of the problem input and may use it to compute a solution. This is called an *offline problem*. In practice, however, uncertainties abound. More often than not only part of the input is known a priori. There are different models that incorporate uncertainties in the input. For example, in stochastic optimization, unknown parts of the input are modeled as random variables following some known probability distribution. This model is often used in preventative maintenance, where the time it takes until a machine or resource breaks down is modeled with an exponential probability distribution, see, e.g., [BAY+09], but also in production scheduling when considering unexpected machine failures, c.f. [FAP96; FAP98]. As production facilities become increasingly interconnected and digitalized, more and more data are being collected allowing for the estimation of underlying probability distributions. See [PFA+20; BHS+22] for a discussion of management and types of uncertainties in production scheduling in the context of Industry 4.0 and Industry 5.0.

Another type of uncertainty is *online information*, where elements of the input appear sequentially, either one by one or over time, and as an element is revealed, an *online algorithm* has to make irreversible decisions about how to handle it without knowledge of the future. This kind of information arises in a variety of settings as many events in the real world are unforeseen and require immediate action, for instance, unexpected customer orders. The online setting is well-researched for many classical optimization problems and has recently also been considered in the context of production scheduling [GM19;

GM20; HBW+21], although rarely when resource constraints or maintenance considerations are present. For a thorough introduction to the topic of online algorithms, we refer the reader to [BE98].

In the hot rolling mill, both types of uncertainties arise: online information in the shape of unforeseen production jobs, and stochastic information in the shape of uncertain processing times and possible machine or resource failures. This combined setting is far less investigated than stochastic and online problems individually, but some research exists, for instance, on how to solve scheduling problems where jobs appear online and have stochastic processing times [MUV06]. However, research has shown that when the level of uncertainty is relatively low, a predictive schedule that is based on offline methods and is recomputed as stochastic information is revealed can outperform online methods [ALM+05]. This is the case for the uncertain processing times in the hot rolling mill as rather good estimates are available in practice, so we ignore these uncertainties and use the approach of frequent replanning. As machine or resource failures are relatively rare, we do not consider them explicitly in our algorithm design and handle them by executing a complete reinitialization. We refer the reader to the review article by Iglesias et al. [EBO+19] for an overview over the different kinds of uncertainties arising in steel manufacturing and existing techniques for dealing with them.

In summary, our work differs from previous research by investigating how to simultaneously find a solution for four aspects of the production process, namely production scheduling, maintenance scheduling, resource constraints, and job-resource allocation, all while handling uncertainties in the production sequence. The bi-criteria objective and a maintenance machine environment in which setup times are incurred when changing the resource type further add to the complexity of our problem. While we are not aware of hardness results for our particular machine environment, Monma and Potts [MP89] showed that such setup times imply $\mathcal{NP}$-hardness when scheduling on two identical parallel machines.

Problem Overview We now turn our attention to the concrete application of this work which comes from the hot rolling mill, a part of a steel plant that is of major importance. Here, slabs of heated steel are rolled between consecutive

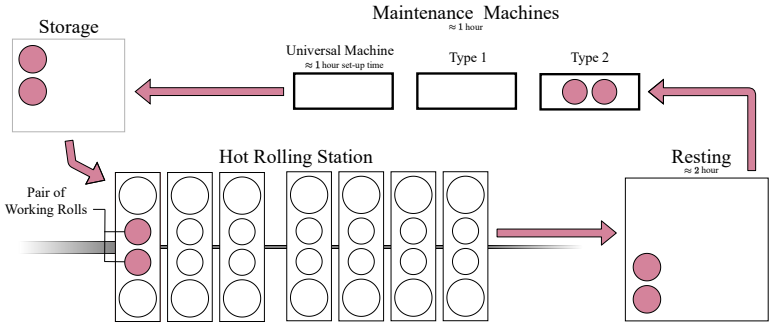


Fig. 4.1: Schematic diagram of the maintenance workflow for working rolls. Starting at the bottom left, at the hot rolling station, pairs of working rolls are used to roll steel slabs into thin sheets. The pair then rests for one hour, before it can be maintained by one of three maintenance machines. There is one type-specific machine for each type plus one universal machine that requires an hour-long setup when switching types. After maintenance the working roll pair can be stored or directly used again at the hot rolling station.

pairs of working rolls to obtain thin steel sheets that are then wound up into coils, with different types and profiles of working rolls determining the properties of the final product. Naturally, such a production process evokes a considerable amount of mechanical stress, that wears down the working rolls and causes a recurring need for their maintenance. After cooling down during a resting period, worn down working rolls can be maintained at a few dedicated machines in order to be used again in future production jobs. Additionally, it is possible to change the profile of a working roll during maintenance. However, depending on the type of a working roll, it can only be maintained by a subset of the machines. A schematic overview of the hot rolling mill maintenance workflow is depicted in Figure 4.1.

The majority of scientific articles regarding the hot rolling mill considers the question of scheduling the processing of slabs and assigning slabs to working roll profiles to match a list of customer orders while satisfying various technical constraints, but does not explicitly take into account the maintenance of working rolls, c.f. [ÖUH21; CWT14; NRJ+05; CYW08; ZSW19] and references therein. However, as the total number of working rolls is limited, their timely maintenance is of utmost importance to the production process as a whole. Failure to produce maintained working rolls with the correct profile when needed at the rolling station can cause

costly delays of the whole milling operation, with the cost of a delay of just one minute ranging in the thousands of euros. At a time when steel manufacturers have to cope with fierce global competition, rising energy prices, a reorientation towards carbon-neutral production, and a decline in demand from the automotive and mechanical industries such delays are an issue of both economical and environmental sustainability to be avoided at all costs, c.f. [VGG+20; Are19; Ryn08; BFC+20b; SEW20; VGG+20].

A second issue with the maintenance of working rolls lies in the assignment of a new profile, which is dictated by the allocation of a worn working roll to an upcoming resource demand with that profile. Since the new profile is ground into the working roll by the maintenance machine, the more a newly chosen profile differs from the previous profile of the working roll, and this leads to a larger loss of working roll mass during maintenance and thus a shorter lifetime. Ideally, one wishes to minimize the overall loss of material among all working rolls and, at the same time, to minimize the idle time of the production schedule, i.e., to cause as little delay from late maintenance as possible. These two goals are usually opposing, and a trade-off that considers the total incurred cost is sought. In practice, however, the cost of delays usually significantly outweighs or even strictly dominates the maintenance cost incurred due to material loss which is reflected in our algorithm design.

In this chapter, we introduce the *simultaneous allocation and maintenance scheduling* (SAMS) problem. Here, we are given a list of n production jobs with associated processing times and *resource demands*, by which we refer to the amount of *resources*, i.e., working roll pairs, of a certain type and profile that a job requires for processing. Further, we are given a set of heterogeneous resources with different types and initial profiles and a set of maintenance machines. The task is to compute feasible schedules for the production jobs on a single processor and for maintenance of resources on the maintenance machines as well as an allocation of resources to resource demands. The objective is to minimizes a cost function that combines the maintenance cost and idle time of the production processor. In SAMS with *lookahead* $k \in \mathbb{N}$, we assume that the list of production jobs is only revealed over time. That is, at any point in time, in addition to the information on production jobs that are being or have already been processed, only the next k upcoming production

	Running Time	Solution Quality	Description
Algorithm 4.1	+++	+	3-stage heuristic using matching and scheduling methods
ILP 1	−−−	+++	general ILP for SAMS
ILP 2	−−	+++	ILP minimizing the idle time
ILP 3	−−	+++	ILP minimizing maintenance cost given production schedule
ILP rolling	−	++	rolling application of an above ILP to handle lookahead k

Tab. 4.1: Overview of the solution methods for SAMS presented in this chapter and their performance with respect to running time and solution quality.

jobs are known. Besides the online nature of the problem, another difficulty lies in the specific machine environment considered for resource maintenance with two kinds of machines. The first kind can only maintain resources of a single type, while the second, the *universal machines*, can maintain resources of any type, but incurs a fixed setup time when switching between types. The complete, formal definition of the SAMS problem is given in Section 4.1.

Our Results Motivated by a project with a large multinational steel manufacturer, the main result in this work is the development of an efficient heuristic for SAMS with lookahead k that produces solutions of good quality while dealing with the various uncertainties that arise in the complex production environment of a hot rolling mill. A fast running time is of particular importance to be able to react to unforeseen changes and (re-)compute a production schedule at a moments notice. We meet this goal by presenting a tunable, event-driven heuristic, Algorithm 4.1, that achieves extremely fast running times and still produces solutions of good quality, particularly with respect to the idle time of the production processor. To rigorously evaluate this heuristic, we also develop several integer linear programs (ILP) for the Offline-SAMS-Problem, ILPs 1-3, that serve as lower bounds to an optimal (offline) solution and compare them to the heuristic on several randomly generated instances that mimic real-world scenarios. Additional experiments show a large improvement when comparing the idle time of the heuristic to that of current real-world schedules created without computational assistance. Table 4.1 gives a summary of the different approaches presented here. The structure of this chapter is as follows.

In Section 4.1, we give a formal definition of the SAMS problem.

After that, in Section 4.2, we present a time-indexed integer linear program, ILP 1, to solve the offline version of SAMS. Additionally, we provide two simpler ILPs that serve as lower bounds to the individual optimization objectives, i.e., waiting cost and maintenance cost. As a consequence, they have considerably faster running times and can be used on some of the instances where the previous, complete ILP is intractable. The first, ILP 2, computes a production schedule with minimum makespan and completely disregards the costs associated with maintenance. The second, ILP 3, given a feasible schedule of production jobs as input, computes an optimal allocation of working rolls to resource demands as well as a feasible maintenance schedule. Since the cost of maintenance is crucially dependent on the underlying production schedule, ILP 3 is particularly useful to evaluate the performance of an algorithm in terms of maintenance cost, by using the schedule computed by the algorithm as input for ILP 3 and then comparing the respective maintenance costs. A comparison with the maintenance cost of an optimal (offline) solution would be misleading since it might have a schedule with shorter makespan, which can constrict maintenance and lead to a larger maintenance cost despite being the better solution overall. We also discuss extensions of the above ILPs to obtain ILP-based solution methods for SAMS with lookahead k.

In Section 4.3, we present Algorithm 4.1, our heuristic for SAMS with lookahead k, which combines fast running times with a good solution quality. It does so by dividing the SAMS problem, with all its complicated and interdependent parts, into three phases, namely resource allocation, maintenance scheduling and production scheduling. These phases are carried out successively and during the execution of each phase, special care is taken that the subsequent phases are not adversely affected. Notably, the resource allocation phase consists of solving a minimum-cost bipartite matching problem, that is cleverly setup in a way that considers the urgency of a resource demand as well as an estimate of the available maintenance capacity. Further, it exploits the observation, that right after a production job j ends and the subsequent resting period finishes, we know that worn working rolls that correspond to the resource demands of j become available for processing. Such *virtual resources* are also incorporated in the minimum-cost matching in an effort to cope with the online nature of the problem.

Section 4.4 presents the results of our experimental evaluation. Here, we compare the aforementioned approaches on randomly generated instances resembling real world-data, to evaluate their performance and highlight the trade-off between running time and solution quality. We further show that the heuristic compares favorably against schedules that are currently used in practice and demonstrate the influence of the tunable parameters within the heuristic.

4.1 Problem Definition

We give a formal definition of the *simultaneous allocation and maintenance scheduling* (SAMS) problem. In the following, when talking about an event that happens at time t and takes x time units, we mean that this event takes place in the time interval $[t, t + x)$. Moreover, for a positive integer n, we denote by $[n]$ the set $\{1, 2, \ldots, n\}$.

In the SAMS problem, we are given a set $J = [n]$ of n production jobs with processing times $p_j \in \mathbb{Q}$. The jobs need to be executed consecutively, in ascending order, on a single machine, the production processor. Moreover, we are given a set R of recyclable resources r, each of which has a type $\alpha_r \in [2]$, an initial profile $\bar{\beta}_r \in B$, where $B \subseteq \mathbb{Q}$ denotes the set of all possible profiles, and an initial state $\gamma \in \Gamma = \{\text{worn, maintained}\}$. For each job $j \in J$ there are associated resource demands $D_{j,\alpha,\beta} \in \mathbb{N}$, for all $\alpha \in [2], \beta \in B$, meaning that to execute job j, we need to expend $D_{j,\alpha,\beta}$ maintained resources with type α and profile β. After processing j, the expended resources change their state to worn. Additionally, there is a resting time of $\Delta_{\mathrm{D}} \in \mathbb{Q}$ additional time units before the resources are available for maintenance. Denote by C_w the total idle time of the processor, to which we also refer to as *waiting cost*.

After resting, a worn resource becomes *available for maintenance*. The maintenance can by carried out by one of three maintenance machines, we call this a *maintenance job*. Maintenance jobs have a processing time of $\Delta_{\mathrm{M}} \in \mathbb{Q}$ time units and at their completion, the state of the corresponding resource is set to maintained. Optionally, the resource may also change its profile, incurring a *maintenance cost* of $|\beta - \beta'|$ when changing the profile from β to β'. Already maintained resources may also change their profile this way. Denote

by C_m the total maintenance cost, i.e., the sum of costs over all maintenance jobs. Of the three machines, machine 1 may maintain only resources of type 1 and machine 2 only those of type 2. The remaining so-called *universal machine*, machine 3, may maintain resources of both types but incurs a setup time of $\Delta_S \in \mathbb{Q}$ time units between maintaining resources of different types during which it cannot process any maintenance jobs. Initially, the universal machine may maintain jobs of type 1 without incurring a setup time.

The task is to obtain valid schedules for the maintenance machines and the production processor together with an allocation of resources to production jobs in order to minimize the combined cost $C_w + \mu \cdot C_m$ for some non-negative parameter μ. Usually, the waiting cost dominates the maintenance cost, i.e., $C_w \gg \mu \cdot C_m$. Note that the assignment of resources to production jobs dictates the profile assignment during the maintenance jobs.

In practice, the production sequence is often not known a priori but is only revealed over time. In the *online SAMS problem with lookahead* k, a production job j, its processing time, and its resource demands are revealed when job $j - k$ starts to be processed; that is, they are known from the beginning if $j - k \leq 0$. Previous decisions, i.e., start of a production or maintenance job and allocation of the corresponding resources, can not be changed after receiving the new information. When $k \geq n$, we have complete information about the instance from the very beginning and are in the offline setting.

When dealing with such uncertainties, we cannot expect to obtain solutions as good as an *offline solution* (also *offline optimum*), an optimal solution for the corresponding offline problem. To evaluate the performance of algorithms under uncertainties, we compare their solutions to an offline solution ($k = n$) for the same instances. Ideally, one would like to bound the ratio of the two for all instances.

We note that, theoretically, it is possible to consider more general versions of the SAMS problem, e.g., with more than two types of resources, arbitrary types and numbers of maintenance machines or different, possibly type-dependent, setup and maintenance times. Motivated by the application in steel production, we consider the specific variant described above. However, the ideas and methods described in this chapter should also be applicable for other variants.

4.2 Integer Linear Programs and Lower Bounds for SAMS

In this section, we present an integer linear program (ILP) for the offline SAMS problem, as well as two additional ILPs that serve as lower bounds for the individual objectives. To this end, we assume that the processing, resting, maintenance, and setup times are all integers. We index time with integer time points t and assume that we are given an upper bound $t_{\max}$ on the makespan of an optimal solution. The time horizon considered by the ILPs is then $T = \{1, \dots, t_{\max}\}$ with values $t \leq 0$ corresponding to the initial state of the problem.

4.2.1 An Integer Linear Programming Formulation for SAMS

We start with the general ILP formulation of SAMS to which we refer from here on as ILP 1. ILP 1 uses binary decision variables $s_{j,t}$ that indicate whether production job $j \in J$ starts being processed at time $t \in T$ and variables $y_{\alpha,t}$ that indicate whether the universal machine can serve resources of type $\alpha \in [2]$ at time $t \in T$. Additionally, we use two types of non-negative integer decision variables: variables $n_{\gamma,\alpha,\beta,t}$ that count the number of resources that have state $\gamma \in \Gamma$, type $\alpha \in [2]$, profile $\beta \in B$ and are available at time $t \in T$, where being available means that the resource is not being used in a maintenance or production job at time t, nor is it resting; and variables $x_{\alpha,\beta,\beta',\gamma,t}$ that count the number of maintenance jobs that are started at time t and change a resource with state γ and type α from profile β to β'. Denote by $\bar{n}_{\gamma,\alpha,\beta}$ the initial number of (available) resources with state $\gamma \in \Gamma$, type $\alpha \in [2]$, and profile $\beta \in B$. For the sake of readability, we will refer with a state $\gamma = 0$ to worn resources and with $\gamma = 1$ to maintained ones. A summary of the indices, sets, parameters and decision variables used in ILP 1 can be found in Table 4.2.

The linear program ILP 1 is formulated as follows.

Indices	
j	production jobs
α	resource types
β, β'	resource profiles
γ	state of resource
t, t'	time points
Sets and Parameters	
J	set $[n]$ of production jobs
B	set of resource profiles
T	set $[t_{\max}]$ of time points
Γ	set of states $\{0, 1\}$; 0 for worn, 1 for maintained
p_j	processing time of job j
$\bar{n}_{\gamma,\alpha,\beta}$	initial number of resources with type α, profile β, and state γ
$D_{j,\alpha,\beta}$	number of resources with type α and profile β needed for job j
Δ_{M}	maintenance time
Δ_{D}	resting time
Δ_{S}	setup time
μ	waiting and maintenance cost trade-off
Decision Variables	
$s_{j,t}$	1, if starting time of job j equals t, 0 otherwise
$y_{\alpha,t}$	1, if univ. machine can serve resources of type α at time t, 0 otherwise
$n_{\gamma,\alpha,\beta,t}$	number of avail. resources of state γ, type α, and profile β at time t
$x_{\alpha,\beta,\beta',\gamma,t}$	number of maintenance jobs starting at time t that change a resource with type α and state γ from profile β to β'

Tab. 4.2: Overview of indices, sets, parameters and decision variables used in ILP 1.

Objective The objective in the SAMS problem is to minimize $C_w + \mu \cdot C_m$, which we express as

$$\min \sum_{t \in T} t \cdot s_{n,t} - \sum_{j=1}^{n-1} p_j + \mu \cdot \sum_{\alpha \in [2]} \sum_{\beta,\beta' \in B} \sum_{\gamma \in \Gamma} \sum_{t \in T} x_{\alpha,\beta,\beta',\gamma,t} \cdot |\beta - \beta'|.$$

Constraints We ensure that every production job is started exactly once and only after the preceding job has been completed:

$$\sum_{t \in T} s_{j,t} = 1, \qquad \text{for all } j \in J,$$

$$s_{j,t} \leq \sum_{t'=1}^{t-p_{j-1}} s_{j-1,t'}, \qquad \text{for all } j \in J \setminus \{1\}, t \in T.$$

The universal machine can maintain only resources of one type at a time and we enforce a setup time of Δ_S between the maintenance of resources with different types, during which $y_{\alpha,t} = 0$:

$$y_{\alpha,t} + \tfrac{1}{1+\Delta_S} \sum_{t'=t-\Delta_S}^{t} y_{(3-\alpha),t'} \leq 1, \qquad \text{for all } \alpha \in [2], t \in T.$$

For all $t \in T$, the number of maintenance jobs being processed per type is at most the current capacity, which is 1 for the type-specific machine plus $y_{\alpha,t}$ for the universal machine:

$$\sum_{t'=t-\Delta_M+1}^{t} \sum_{\beta,\beta' \in B} \sum_{\gamma \in \Gamma} x_{\alpha,\beta,\beta',\gamma,t} \leq 1 + y_{\alpha,t}, \quad \text{for all } \alpha \in [2], t \in T.$$

To track the number of available resources, note that the number of maintained resources of type α and profile β increases only when an appropriate maintenance job with target profile β is completed. It decreases by one whenever a maintenance job of a maintained resource with source profile β is started or when a production job j starts its processing, in which case it decreases by precisely $D_{j,\alpha,\beta}$. Similarly, the number of worn resources of a certain type and profile decreases when a maintenance job with that type and source profile is started, and it increases after a production job is finished and the subsequent resting time is elapsed:

$$n_{1,\alpha,\beta,t} - n_{1,\alpha,\beta,t-1} = \sum_{\beta' \in B} \Big(\sum_{\gamma \in \Gamma} x_{\alpha,\beta',\beta,\gamma,t-\Delta_M} - x_{\alpha,\beta,\beta',1,t} \Big) - \sum_{j \in J} D_{j,\alpha,\beta} \cdot s_{j,t},$$
$$\text{for all } \alpha \in [2], \beta \in B, t \in T,$$

$$n_{0,\alpha,\beta,t} - n_{0,\alpha,\beta,t-1} = - \sum_{\beta' \in B} x_{\alpha,\beta,\beta',0,t} + \sum_{j \in J} D_{j,\alpha,\beta} \cdot s_{j,t-p_j-\Delta_D},$$
$$\text{for all } \alpha \in [2], \beta \in B, t \in T.$$

Note that, since the variables $n_{\gamma,\alpha,\beta,t}$ are non-negative, the above equations also ensure that when a production or maintenance job is started the required amount and variety of resources is available.

Finally, we set the initial values as

$$y_{1,t} = 1, \quad n_{\gamma,\alpha,\beta,t} = \bar{n}_{\gamma,\alpha,\beta}, \quad s_{j,t} = 0, \quad x_{\alpha,\beta,\beta',\gamma,t} = 0, \\ \text{for all } \alpha \in [2], \beta, \beta' \in B, \gamma \in \Gamma, t \leq 0,$$

and require the decision variables to be non-negative integers

$$s_{j,t}, y_{\alpha,t}, n_{\gamma,\alpha,\beta,t}, x_{\alpha,\beta,\beta',\gamma,t} \in \mathbb{N}_0, \quad \text{for all } \alpha \in [2], \beta, \beta' \in B, \gamma \in \Gamma, j \in J, t \in T.$$

While ILP 1 is guaranteed to output an optimal solution, its running time, even for moderate step sizes of the time indexing, is prohibitively large or even intractable. Hence, the main use case of ILP 1 lies in the evaluation of more practical algorithms or in dealing with longer-term questions like considerations regarding the production environment. For instance, we can utilize ILP 1 to answer questions such as "what improvements can I possibly expect to see when adding an additional maintenance machine?", where the time it takes to compute the solution is of minor importance.

4.2.2 Lower Bounding the Individual Objectives

To evaluate the performance of algorithms for SAMS, we would like to compare their objective value with that of an optimal offline solution OPT. However, solving ILP 1 is not tractable even for problems of moderate size. We present two simpler ILPs to evaluate the performance of an algorithm solely with respect to waiting and maintenance cost, respectively. The first, ILP 2, computes a schedule with minimum idle time while ensuring that all resources are in fact maintained when they are expended by a processing job but without taking into account the actual maintenance costs. The second, ILP 3, takes as input a feasible schedule of production jobs, and computes the assignment of resources to type-profile demands, as well as a feasible maintenance machine schedule that minimizes the maintenance cost while respecting the given production schedule. To judge the quality of an algorithm with respect to maintenance cost, we use its production schedule as input for ILP 3 to obtain the minimum waiting cost that is achievable when given this fixed schedule.

Recalling that in practice the waiting cost usually dominates the maintenance cost, one may think that under such an assumption, it is possible to exactly compute OPT by first executing ILP 1 and subsequently ILP 2, with the previously computed optimal production schedule as input. Unfortunately, this is not the case as there may exist several different schedules of minimum waiting cost. With the temporal constraints imposed by these schedules on the maintenance of resources, the minimum maintenance costs for these schedules may vary greatly. To find OPT we would thus need to compute all such schedules and apply ILP 2 to each of them.

ILP 2 - Lower Bounding the Idle Time We describe a simplification of ILP 1, which we call ILP 2, that computes (offline) solutions with minimum idle time, but disregards maintenance costs. As in ILP 1, we use binary decision variables $s_{j,t}$ that indicate whether production job $j \in J$ starts being processed at time $t \in T$ and variables $y_{\alpha,t}$ that indicate whether the universal machine can maintain resources of type $\alpha \in [2]$ at time $t \in T$. Additionally, we use two types of non-negative integer decision variables: variables $n_{\gamma,\alpha,t}$ that count the number of resources that have state $\gamma \in \Gamma$, type $\alpha \in [2]$, and are available at time $t \in T$, where being available means that the resource is not being used in a maintenance or production job at time t, nor is it resting; and variables $x_{\alpha,\gamma,t}$ that count the number of maintenance jobs that are started at time t on worn resources of type α. Denote by $\bar{n}_{\gamma,\alpha}$ the initial number of (available) resources with state $\gamma \in \Gamma$ and type $\alpha \in [2]$. Again, we refer with a state $\gamma = 0$ to worn resources and with $\gamma = 1$ to maintained ones and summarize the indices, sets, parameters and decision variables, see Table 4.3.

The linear program ILP 2 is formulated as follows.

Objective The objective of ILP 2 is to minimize the idle time only, that is,

$$\min \sum_{t \in T} t \cdot s_{n,t} - \sum_{j=1}^{n-1} p_j.$$

Indices	
j	production jobs
α	resource types
γ	state of resource
t, t'	time points
Sets and Parameters	
J	set $[n]$ of production jobs
T	set $[t_{\max}]$ of time points
Γ	set of states $\{0, 1\}$; 0 for worn, 1 for maintained
p_j	processing time of job j
$\bar{n}_{\gamma,\alpha}$	initial number of resources with type α and state γ
$D_{j,\alpha}$	number of maintained resources with type α needed for job j
Δ_{M}	maintenance time
Δ_{D}	resting time
Δ_{S}	setup time
Decision Variables	
$s_{j,t}$	1, if starting time of job j equals t, 0 otherwise
$y_{\alpha,t}$	1, if universal machine can serve resources of type α at time t, 0 otherwise
$n_{\gamma,\alpha,t}$	number of available resources with state γ and type α at time t

Tab. 4.3: Overview of indices, sets, parameters and decision variables used in ILP 2.

Constraints We ensure that every production job is started exactly once and only after the preceding job has been completed:

$$\sum_{t \in T} s_{j,t} = 1, \qquad \text{for all } j \in J,$$

$$s_{j,t} \leq \sum_{t'=1}^{t-p_{j-1}} s_{j-1,t'}, \qquad \text{for all } j \in J \setminus \{1\}, t \in T.$$

The universal machine can maintain only resources of one type at a time and we enforce a setup time of Δ_{S} between the maintenance of resources with different types, during which $y_{\alpha,t} = 0$:

$$y_{\alpha,t} + \tfrac{1}{1+\Delta_{\mathrm{S}}} \sum_{t'=t-\Delta_{\mathrm{S}}}^{t} y_{(3-\alpha),t'} \leq 1, \qquad \text{for all } \alpha \in [2], t \in T.$$

For all $t \in T$, the number of maintenance jobs being processed per type is at most the current capacity, which is 1 for the type specific machine plus $y_{\alpha,t}$ for the universal machine:

$$\sum_{t'=t-\Delta_{\mathrm{M}}+1}^{t} \sum_{\gamma \in \Gamma} x_{\alpha,\gamma,t} \leq 1 + y_{\alpha,t}, \qquad \text{for all } \alpha \in [2], t \in T.$$

To track the number of available resources, note that the number of maintained resources of type α increases by one whenever a maintenance job with that type is completed and decreases when a production job j starts its processing, in which case it decreases by precisely $D_{j,\alpha}$. Similarly, the number of worn resources of a certain type decreases when a maintenance job with that type is started, and it increases after a production job is finished and the subsequent resting time is elapsed:

$$n_{1,\alpha,t} - n_{1,\alpha,t-1} = \quad x_{\alpha,1,t} - \sum_{j \in J} D_{j,\alpha} \cdot s_{j,t}, \qquad \text{for all } \alpha \in [2], t \in T,$$

$$n_{0,\alpha,t} - n_{0,\alpha,t-1} = -x_{\alpha,0,t} + \sum_{j \in J} D_{j,\alpha} \cdot s_{j,t-p_j-\Delta_{\mathrm{D}}}, \qquad \text{for all } \alpha \in [2], \beta \in B, t \in T.$$

Note that, since the variables $n_{\gamma,\alpha,t}$ are non-negative, the above equations also ensure that when a production or maintenance job is started the required amount of maintained resources with correct type is available.

Finally, we set the initial values as

$$y_{1,t} = 1,\ n_{\gamma,\alpha,t} = \bar{n}_{\gamma,\alpha},\ s_{j,t} = 0,\ x_{\alpha,\gamma,t} = 0, \quad \text{for all } \alpha \in [2], \gamma \in \Gamma, t \leq 0,$$

and require the decision variables to be non-negative integers

$$s_{j,t}, y_{\alpha,t}, n_{\gamma,\alpha,t}, x_{\alpha,\gamma,t} \in \mathbb{N}_0, \qquad \text{for all } \alpha \in [2], \gamma \in \Gamma, j \in J, t \in T.$$

ILP 3 - Lower Bounding the Allocation Cost of a Feasible Schedule To complement ILP 2, we need another integer linear program for SAMS that computes an optimal allocation of resources to resource demands when given a feasible schedule for the production processor as an input. For this, we may

simply use ILP 1 and fix the starting times of production jobs according to that schedule. We refer to this linear program as ILP 3.

4.2.3 Using ILPs for SAMS with Lookahead k

All three ILPs described in this section compute optimal offline solutions. As such, they provide lower bounds that we cannot expect to reached when considering SAMS with lookahead k. However, it is actually possible to utilize such an ILP in the setting of constant lookahead k. We do this by repeatedly applying the ILP, in a rolling manner. That is, whenever a new production job is revealed, recompute an updated schedule and allocation for the next k production jobs using ILP 1, with previous, irrevocable decisions fixed as initial values. We refer to this method as *ILP rolling*, e.g., ILP 1 rolling for the rolling application of ILP 1. While the obtained solutions are no longer optimal, they are still of good quality. However, despite the resulting small instances with only k production jobs, the time-indexing of the ILPs causes running times to be too high still for step sizes that suit practical needs.

4.3 An Efficient Heuristic

Since we are able to compute optimal (offline) solutions using ILP 1, a natural approach to solving SAMS with lookahead k would be to repeatedly apply ILP 1 in a rolling manner, as described in the previous section. Unfortunately, the assumption of having integer time points and duration parameters, i.e., processing times, maintenance duration, etc., presents a problem. Choosing a sufficiently small stepsize to properly reflect real-world behavior, dramatically increases the complexity of the ILP, rendering the computation of a solution excessively slow or even completely intractable, as we show in our experimental evaluations in Section 4.4. One could also round up the various duration parameters to a multiple of a chosen, possibly larger, stepsize, but this in turn could greatly increases the obtained makespan. These downsides are not tolerable as even small production delays are extremely costly and schedules are often required to be (re)computed at a moments notice to react to unforeseen changes due to the manifold uncertainties associated

with real-world production processes. Although ILP 2 and ILP 3 are much simpler than ILP 1, they cause the same problems as described above and are not tractable when dealing with small stepsizes.

In this section, we present a heuristic for SAMS that efficiently computes solutions of good quality while handling the various uncertainties associated with real-world production processes. Besides unknown production sequence with a bounded lookahead, we differentiate between two types of uncertainty. First, minor, expected uncertainties that frequently occur and manifest as stochastic duration parameters; these are handled during the normal execution of the heuristic by frequently recomputing the maintenance schedule whenever new information becomes available. And second, large unexpected changes such as the removal or addition of production jobs or resources which require a complete reinitialization of the heuristic.

For a job j, we denote by s_j the *planned starting time* of the job, i.e., the earliest possible starting time assuming that all resource demands of the job and its predecessors are met. Note that this value may change dynamically over time whenever delays occur due to unmet resource demands. Moreover, we consider the resource demands of job j as several individual *type-profile demands*, namely $D_{j,\alpha,\beta}$ demands τ with type $\alpha = \alpha_\tau$ and profile $\beta = \beta_\tau$, and denote by D_j the set of all type-profile demands associated with job j. An important observation we use is the fact that if job j is processed at time t, then $|D_j|$ worn resources r will become available for maintenance at time $a_r := t + p_j + \Delta_{\mathrm{D}}$, each of them is associated, one to one, with a demand $\tau = \tau(r) \in D_j$ and has the corresponding type and profile. We call these *virtual resources* and denote by r_τ the virtual resource associated with τ. Our heuristic takes such resources into account when computing an assignment of (virtual) resources to type-profile demands in order to cope with the limited lookahead when minimizing the maintenance cost.

On a high level, the heuristic consists of three phases that are executed consecutively, whenever new information becomes available. Each of the phases is designed in a way that benefits the subsequent phases. In the first phase, the allocation phase, we express the assignment of (virtual) resources to type-profile demands as a minimum-cost matching problem. Its cost function not only reflects the goal to minimize the maintenance cost, but also tries to ensure that resources are assigned primarily to production

jobs with a sooner starting time and that there is still sufficient time for their maintenance. For the initial assignment, we use the same approach but separately consider maintained available resources and resources that are being maintained and assign them to the most imminent production jobs. The second phase, the maintenance scheduling phase, schedules maintenance jobs that are induced by the allocation from resources to demands from the first phase on the three maintenance machines. The planned starting time of a production job acts as a due date for the associated maintenance jobs and we schedule according to the earliest due date (EDD) rule, which is known to be optimal if the amount of setup actions and their precise times are known [MP89, Theorem 1]. For the universal machine, we try various set-up times and pick the best option. Finally, in the third phase, we simply schedule the next production job as soon as the required resources become available.

Algorithm Description After the initial resource allocation, which we describe further below, the normal execution of the heuristic consists of three phases that are executed repeatedly whenever new information is revealed. Specifically, whenever a process finishes, or a resources become available for maintenance, or new information is revealed, execute the following steps, summarized as Algorithm 4.1.

Phase I: Allocation We consider the weighted bipartite graph $G = (U \cup W, E, c)$ that is defined as follows. The set U consists of all resources that are currently available for maintenance and all known virtual resources that will become available for maintenance in the future. The set W contains all type-profile demands that are not *satisfied*, i.e., there is no resource of the correct type and profile that is maintained or currently in maintenance allocated to this demand yet. Specifically, resources and type-profile demands allocated in previous iterations for which the associated maintenance job has not been started yet are considered anew and their former allocation is discarded. A (virtual) resource r is connected to a type-profile demand τ with an edge of E, if we have $\alpha_r = \alpha_\tau$ and if the inequality $\mu_c \cdot v_\tau \leq s_\tau - a_r$ holds, where $\mu_c \in \mathbb{Q}$ is a tunable parameter and $v_\tau = \Delta_M \cdot |\{\tau' \in D \mid \tau' \text{ not satisfied and } t_{now} \leq s_{\tau'} \leq s_\tau\}|$ is the volume of maintenance jobs that we aim to finish before time s_τ, with t_{now} denoting the current time. Intuitively, fulfilling this inequality means, that we are

Algorithm 4.1 Regular Operation at Time t_{now}

Phase I: Allocation

consider the weighted bipartite graph $G = (U \cup W, E, c)$ with
  $U = \{r \text{ available for maintenance}\} \cup \{r \text{ virtual resource with } a_r > t_{\text{now}}\}$
  $W = \{\tau \mid \tau \text{ not satisfied}\}$
  $E = \{(r, \tau) \mid \text{types match and } \mu_{\text{c}} \cdot v_\tau \leq s_\tau - a_r\}$
  $c(r, \tau) = |\beta_r - \beta_\tau| + m_{\max} \cdot D_{\max} \cdot j(\tau)$
$M \leftarrow$ minimum-cost maximum matching on G

Phase II: Maintenance Scheduling

compute due dates s_j (starting time for job j assuming no idle time)
$\mathcal{S}_0 \leftarrow$ schedule M according to EDF without setup
for $j = 1, \ldots, |M|$ **do**
  $\mathcal{S}_j \leftarrow$ schedule M w.r.t. EDF with setup before j-th job on machine 3
$j^* \leftarrow \arg\min_{j=1,\ldots,|M|} L_{\max}(\mathcal{S}_j)$
if $L_{\max}(\mathcal{S}_0) = 0$ or $L_{\max}(\mathcal{S}_0) \leq L_{\max}(S_{j^*}) + \mu_{\text{p}}$ **then**
  $\mathcal{S} \leftarrow S_0$
else
  $\mathcal{S} \leftarrow \mathcal{S}_{j^*}$
start all possible maintenance jobs according to $\mathcal{S}$ and M

Phase III: Production

if possible, start next production job according to M and previous allocations

confident that the maintenance of r can be accomplished before time s_τ, and this confidence can be adjusted using the parameter μ_{c}. Lastly, the edge weights of the graph are given by $c(r, \tau) = |\beta_r - \beta_\tau| + m_{\max} \cdot D_{\max} \cdot j(\tau)$, where $D_{\max}$ is an upper bound on the number of demands per production job, $m_{\max}$ is an upper bound on the maintenance cost of any job, and $j(\tau)$ is the index of the production job associated with τ. A minimum-cost maximum matching on G with respect to this cost function attains the lowest maintenance cost possible under the constraints that only pairs in E are matched and that more urgent production jobs, i.e., those with an earlier planned starting time, are prioritized over less urgent ones. The latter is achieved by the additive term of $m_{\max} \cdot D_{\max} \cdot j(\tau)$ in the cost function, which ensures that the cost of satisfying less urgent type-profile demands dominates that of more urgent ones. The computation of a minimum-cost maximum matching M on G concludes Phase I.

Phase II: Maintenance Scheduling The second phase consists of scheduling the maintenance jobs obtained from the matching M computed in Phase I. A maintenance job $(r, \tau) \in M$ can start its processing at the earliest at time a_r, when the corresponding resource becomes available for maintenance, and is

ideally finished by time $s_{j(\tau)}$. Thus, the problem of scheduling the resource maintenance presents itself as a scheduling problem with release times a_r and due dates $s_{j(\tau)}$ in a machine environment with setup times in which we want to minimize the maximum lateness $L_{\max}$, i.e., the maximum difference between a job's completion time and its due date. For the two type-specific machines, machines 1 and 2, we always schedule available maintenance jobs according to EDD and prioritize scheduling on these machines over scheduling on the universal machine, since EDD is known to be optimal when no setups occur. To determine whether and when to use a setup on the universal machine, we compute schedules using EDD for the cases that no setup is used and with a single setup for every possible time point in the maintenance horizon, which we define to be just large enough to guarantee the completion of all jobs. From the computed schedules, we pick the one with the best objective value. However, since setups consume valuable machine and personnel time and a practitioner might want to avoid them as much as possible, we add a tunable penalty of $\mu_{\mathrm{p}} \in \mathbb{Q}$ to the objective value of the schedules that use a setup when choosing the maintenance schedule above. We note that while it would be possible to consider more than a single setup, the maintenance horizon is relatively small, and, as the maintenance schedule is recomputed frequently, this gives plenty of opportunity to schedule more setups if needed. For all idle maintenance machines, start a job or setup according to the chosen maintenance schedule.

Phase III: Production If the production processor is idle and all resource demands for the next job are met, i.e., there are maintained resources of the required type and profile, start the next production job.

The above implicitly assumes that maintained resources are either awaiting the start of their allocated production job, are being used therein, or are resting thereafter, and that worn resources are either available for maintenance or allocated to a production job with their maintenance underway or already completed. While this is true under continuous operation, it is not so during initialization or when problem parameters suddenly and unforeseeably change and force a reinitialization. In this case, we need to additionally consider maintained resources or resources currently being maintained that are not allocated to a production job yet. Such resources are assigned with priority in an initial allocation described below as Phase 0.

Phase 0: Initial Allocation To allocate (almost) maintained resources during initialization, we use the same approach as in Phase I. That is, we compute a minimum-cost maximum matching on a weighted bipartite graph $G = (U \cup W, E, c)$. The set U now contains all maintained resources and all worn resources that are in the process of being maintained and the set W consists of the $2 \cdot |U|$ most urgent type-profile demands and those of equal urgency. A resource in U is connected to a demand in W via E, if and only if both, type and profile, already match. For resources that are being maintained, we already know which profile they will have after completing maintenance and this is the profile considered when computing E. The edge weights are given by $c(r, \tau) = b_r + b_{\max} \cdot D_{\max} \cdot j(\tau)$, where b_r denotes the time when r completes maintenance, i.e., $b_r = 0$ for maintained resources, and $b_{\max}$ is an upper bound on b_r. This way, preference is given to more urgent production jobs, and also to earlier maintained resources. After this initial allocation, the three other phases are executed. Maintained resources that were not allocated in Phase 0, are considered in Phase I, for all future iterations, until allocated, together with the remaining resources in the set V. Additionally, all edges to demands τ with the same type and profile have a cost of $c(r, \tau) = m_{\max} \cdot D_{\max} \cdot (j(\tau) - n)$ to encourage their use without renewed maintenance.

4.4 Experimental Results and Interpretation

We present the results of our experimental evaluation of Algorithm 4.1 for SAMS with lookahead k. To reflect the setting in the hot rolling mill, we use the following problem parameters: the lookahead is $k = 4$; the numbers of resources of type 1 and 2 are 15 and 17, respectively; maintenance and setup times equal 1 hour each; and the resting time equals 2 hours. For Algorithm 4.1, we set the tunable parameters to $\mu_c = 1$ and $\mu_p = 1.33$.

Figure 4.2 compares the idle times of Algorithm 4.1 with those of manually created schedules currently used in practice. It does so on twenty instances containing 50 production jobs each that were extracted from real-world data. Clearly, Algorithm 4.1 has a lower idle time on every instance. In fact, in the majority of instances, it beats the real-world schedules by a large margin. This would imply a considerable potential for cost reduction in practice. However,

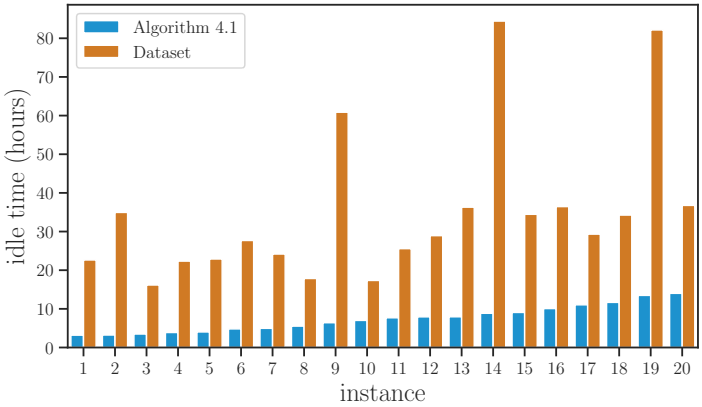
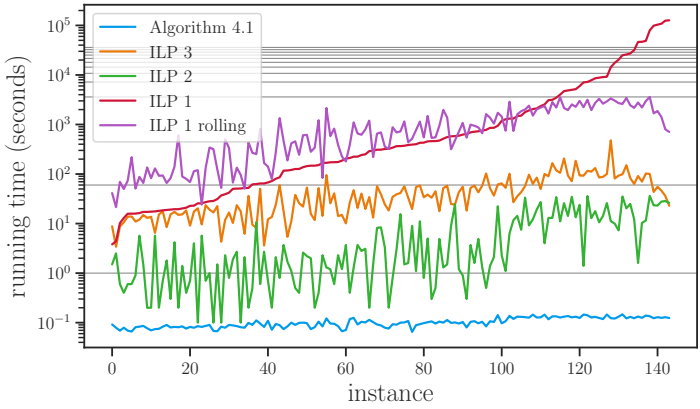


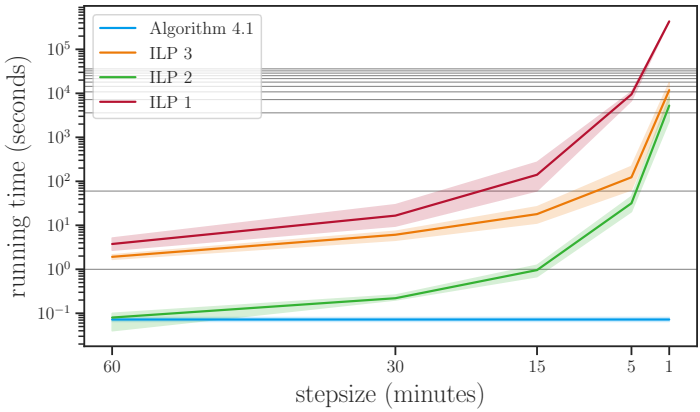
Fig. 4.2: Idle times of Algorithm 4.1 and a dataset for 20 real-world instances. The sorting is according to the idle time for Algorithm 4.1.

such results have to be taken with a grain of salt. Although great care was taken to clean up the raw data, some of the idle time in the real-world schedules may not be due to a worse quality of the schedule computation itself, but due to other factors, for instance machine breakdowns or longer maintenance times, that are unknown to us and not taken into account when running Algorithm 4.1.

For this reason, and to be able to rigorously evaluate the performance of Algorithm 4.1 with respect to maintenance cost and idle time against strict lower bounds, we run further evaluations on artificially generated instances. These were created to resemble real-world datasets while restricting processing times of production jobs to multiples of one hour. Specifically, processing times, in hours, were chosen uniformly at random from the set $\{1, 2, 3\}$. The other parameters, i.e., number of resources and duration parameters are as above. Since all durations are multiples of one hour, we can use integer linear programs from Section 4.2 with 1 hour time steps, rendering ILPs 1-3 tractable for these instances. In the following, we use *ILP 1 rolling* to refer to the rolling application of ILP 1 as discussed in Section 4.2.3. Further, when considering ILP 3, we specifically mean that we use ILP 3 with the schedule obtained by Algorithm 4.1 as input.



(a) Running times of all algorithms on 150 instances.



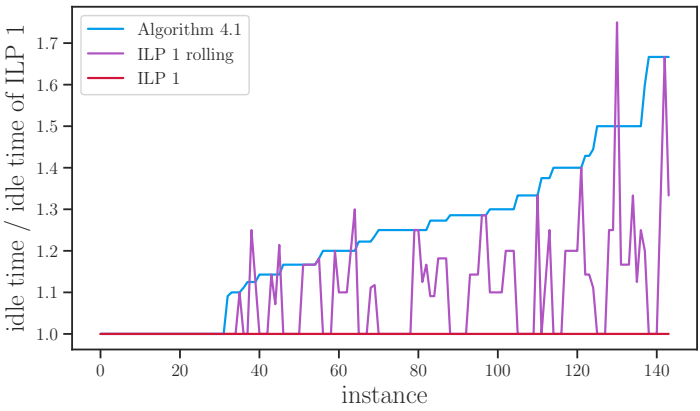
(b) Running times for small stepsizes on instances with 4 programs.

Fig. 4.3: Evaluation of the running time (in seconds) of the different algorithms. The thin gray lines indicate – starting from below – 1 second, 1 minute, and every hour until 10 hours for the topmost line. The shaded, wider lines indicate the 95% confidence intervals.

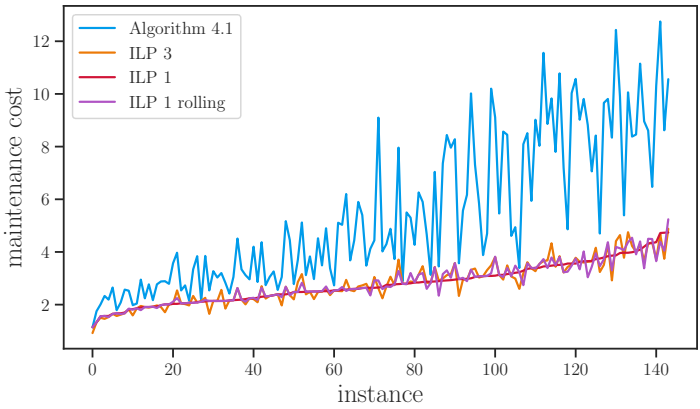
We ran experiments on 150 randomly generated instances as described above, with three groups of 50 instances containing 5, 10, and 20 production jobs each. Figure 4.3a compares the running times of Algorithm 4.1, ILPs 1-3 and ILP 1 rolling. Algorithm 4.1 is clearly the fastest approach, followed by ILP 3, then ILP 2, then ILP 1 rolling, and lastly ILP 1, each time differing

by at least an order of magnitude. To emphasize the influence of the ILP stepsize on the running time, we ran additional experiments on 5 instances with only 4 production jobs for stepsizes 1, 5, 15, 30, and 60 minutes. We stopped the computation on an instance when the running time reached 120 hours. The results are shown in Figure 4.3b. While the running time of the heuristic does not depend on the stepsize, the running times of the different ILPs increase rapidly when the stepsize decreases. ILP 1 reaches the time cap of 120 hours for the 1 minute stepsize on all instances. Since the instances consist of only 4 production jobs, ILP 1 is equivalent to ILP 1 rolling here. This means that a rolling application of ILP 1 is not feasible in practice. Even ILP 2, the fastest of the integer linear programs, has an average running time of 35 seconds and 75 minutes for stepsizes 5 minutes and 1 minute, respectively, while Algorithm 4.1 has an average running time of just 72 milliseconds for arbitrary stepsizes/precision. The solutions of the ILPs were computed with CPLEX in Ubuntu 18.04.5 on a machine with two AMD EPYC ROME 7542 CPUs (64 cores in total, 1 core per CPLEX execution) and 1.96 TB RAM using Python and CPLEX, while Algorithm 4.1 ran on a laptop with a AMD Ryzen 7 PRO 5850U CPU.

We used the same 150 instances as before to further evaluate the performance of Algorithm 4.1. Figure 4.4a shows a comparison of the idle times of Algorithm 4.1, the (offline) optimum OPT (as computed by ILP 1 or ILP 2), and ILP 1 rolling. The values are normalized through division by OPT. On average, the idle times of Algorithm 4.1 and ILP 1 rolling are 23.7% and 8.9% higher than that of OPT, and at most 66.7% and 75% higher, respectively. Given the online nature of the problem, this seems to be a good performance. Turning to the maintenance cost, however, Algorithm 4.1 performs not quite as good when comparing it with the integer linear programs, see Figure 4.4b. Specifically, the maintenance cost of Algorithm 4.1 is by 161% larger than that of ILP 3. We suspect that this is so because our algorithm design focuses primarily on minimizing the idle time, which also is the more important objective in practice. Algorithmic decisions taken in order to lower the idle time may have adversely impacted the maintenance cost. This is nicely illustrated by the maintenance cost of ILP 1 rolling and ILP 3, which for some instances are lower than those of the optimal solution (optimal for the bi-criteria objective!). This seemingly odd behavior is due to the fact that schedules which are worse with respect to idle time leave more room for



(a) Idle time normalized by OPT for Algorithm 4.1 and ILP 1 rolling.

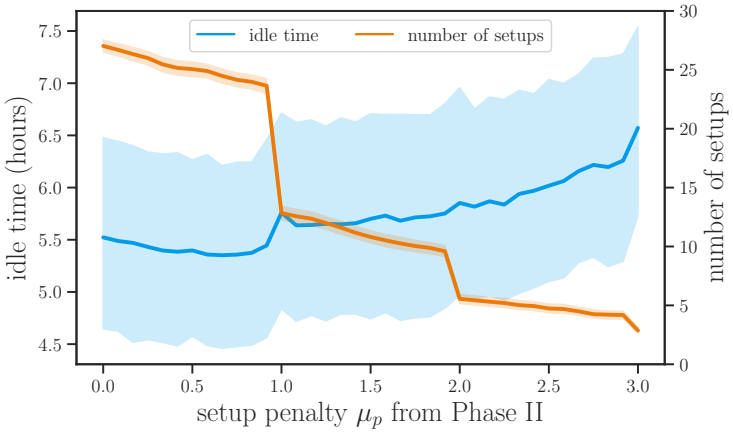


(b) Maintenance cost for Algorithm 4.1, ILP 1, ILP 1 rolling, and ILP 3.

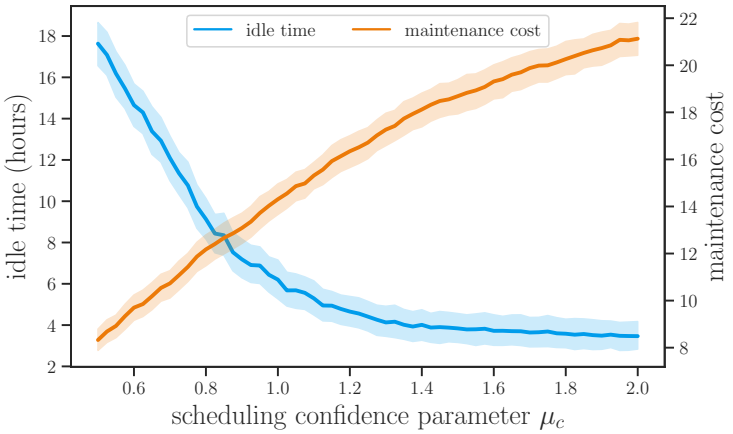
Fig. 4.4: Results of the evaluation for waiting and maintenance cost on 150 instances.

maintenance and thus more flexibility to reduce the maintenance cost. Since ILP 3 works with the starting times of Algorithm 4.1, we know that a better solution is possible, at least in the offline setting. Further, as ILP 1 rolling uses lookahead as well, it seems that in the online setting, too, there is room for improvement.

Figure 4.5 illustrates the influence of the tunable parameters on the performance of Algorithm 4.1. Figure 4.5a shows that the idle time increases with



(a) Influence of the parameter μ_p on idle time and number of setups.



(b) Influence of the parameter μ_c on idle time and maintenance cost.

Fig. 4.5: Influence of the parameters μ_p (setup penalty) and μ_c (scheduling confidence) on the performance of Algorithm 4.1. The shaded, wider lines indicate the 95% confidence intervals.

a larger value of μ_p (setup penalty in Phase II), but only slightly, while the number of setups decreases significantly, especially on multiples of 1 hour. This motivates our choice of $\mu_p = 1.33$. In Figure 4.5b, we see that depending on the choice of μ_c (scheduling confidence parameter in Phase I) there is

a trade-off between idle time and maintenance cost. For small values of μ_c allocations that are beneficial with respect to maintenance cost are chosen even if timely maintenance is unlikely, while large values prioritize allocations where timely maintenance can be guaranteed. This trade-off seems unavoidably but the parameter μ_c offers an opportunity for practitioners to fine-tune it according to their needs.

4.5 Conclusion

In this chapter, motivated by the application of maintaining working rolls in the hot rolling mill of a large international steel manufacturer, we introduced the simultaneous allocation and maintenance scheduling (SAMS) problem. We gave an integer linear programming formulation for SAMS as well as two simpler linear programs that lower bound the individual optimization objectives. They are mostly useful for production design and the evaluation of heuristics. For operational use, we presented a heuristic for SAMS that uses matching and scheduling techniques to obtain solutions for SAMS with lookahead k. Finally, the heuristic was evaluated against real-world data sets and the developed integer linear programs which showed its vastly superior running time and ability to compute good-quality solutions, particularly with respect to the idle time. We hope that this work furthers the investigation of how to deal with different aspects of the production process in an integrated manner.

References

[AAG+19] A. Abboud, R. Addanki, F. Grandoni, D. Panigrahi, and B. Saha. "Dynamic set cover: improved algorithms and lower bounds". In: *STOC*. ACM, 2019, pp. 114–125 (cit. on p. 51).

[AW14] A. Abboud and V. V. Williams. "Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems". In: *FOCS*. IEEE Computer Society, 2014, pp. 434–443 (cit. on p. 51).

[ACW15] A. Adamaszek, P. Chalermsook, and A. Wiese. "How to Tame Rectangles: Solving Independent Set and Coloring of Rectangles via Shrinking". In: *APPROX*. Vol. 40. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 43–60 (cit. on pp. 106, 107, 109).

[AW15] A. Adamaszek and A. Wiese. "A quasi-PTAS for the Two-Dimensional Geometric Knapsack Problem". In: *SODA*. SIAM, 2015, pp. 1491–1505 (cit. on p. 98).

[AW13] A. Adamaszek and A. Wiese. "Approximation Schemes for Maximum Weight Independent Set of Rectangles". In: *FOCS*. IEEE Computer Society, 2013, pp. 400–409 (cit. on p. 98).

[AJS+18] H. A. Akitaya, M. D. Jones, D. Stalfa, and C. D. Tóth. "Maximum Area Axis-Aligned Square Packings". In: *MFCS*. Vol. 117. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 77:1–77:15 (cit. on p. 98).

[AB97] A. Alfieri and P. Brandimarte. "Job shop scheduling with delay-renewable resources: A comparison of decomposition strategies". In: *International Journal of Production Research* 35.7 (1997), pp. 1807–1824. eprint: https://doi.org/10.1080/002075497194921 (cit. on p. 113).

[ADJ18] S. Angelopoulos, C. Dürr, and S. Jin. "Online Maximum Matching with Recourse". In: *MFCS*. Vol. 117. LIPIcs. 2018, 8:1–8:15 (cit. on p. 15).

[AFT18] A. Antoniadis, C. Fischer, and A. Tönnis. "A Collection of Lower Bounds for Online Matching on the Line". In: *LATIN*. Vol. 10807. Lecture Notes in Computer Science. 2018, pp. 52–65 (cit. on pp. 5, 13, 37, 38).

[ABC+19] A. Antoniadis, F. Biermeier, A. Cristi, C. Damerius, R. Hoeksma, D. Kaaser, P. Kling, and L. Nölke. “On the Complexity of Anchored Rectangle Packing”. In: *ESA*. Vol. 144. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 8:1–8:14 (cit. on pp. 7, 97, 109).

[Are19] M. Arens. “Policy support for and R&D activities on digitising the European steel industry”. In: *Resources, Conservation and Recycling* 143 (2019), pp. 244–250 (cit. on p. 116).

[ADN13] C. Artigues, S. Demassey, and E. Néron. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE. Wiley, 2013 (cit. on p. 113).

[AR01] C. Artigues and F. Roubellat. “A Petri net model and a general method for on and off-line multi-resource shop floor scheduling with setup times”. In: *International Journal of Production Economics* 74.1 (2001). Poductive Systems: Strategy, Control, and Management, pp. 63–75 (cit. on p. 113).

[AT19] K. Axiotis and C. Tzamos. “Capacitated Dynamic Programming: Faster Knapsack and Graph Algorithms”. In: *ICALP*. Vol. 132. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 19:1–19:13 (cit. on p. 93).

[ALM+05] H. Aytug, M. A. Lawley, K. McKay, S. Mohan, and R. Uzsoy. “Executing production schedules in the face of uncertainties: A review and some future directions”. In: *European Journal of Operational Research* 161.1 (2005), pp. 86–110 (cit. on p. 114).

[BHS+22] K. Bakon, T. Holczinger, Z. Süle, S. Jaskó, and J. Abonyi. “Scheduling Under Uncertainty for Industry 4.0 and 5.0”. In: *IEEE Access* 10 (2022), pp. 74977–75017 (cit. on p. 113).

[BDT17] K. Balas, A. Dumitrescu, and C. D. Tóth. “Anchored rectangle and square packings”. In: *Discrete Optimization* 26 (2017), pp. 131–162 (cit. on pp. 98, 105).

[BBG+14] N. Bansal, N. Buchbinder, A. Gupta, and J. Naor. “A Randomized $O(\log^2 k)$-Competitive Algorithm for Metric Bipartite Matching”. In: *Algorithmica* 68.2 (2014), pp. 390–403 (cit. on p. 14).

[BK14] N. Bansal and A. Khan. “Improved Approximation Algorithm for Two-Dimensional Bin Packing”. In: *SODA*. SIAM, 2014, pp. 13–25 (cit. on p. 98).

[BH60] R. Barlow and L. Hunter. “Optimum Preventive Maintenance Policies”. In: *Operations Research* 8.1 (1960), pp. 90–100. eprint: https://doi.org/10.1287/opre.8.1.90 (cit. on p. 111).

[BP96] R. E. Barlow and F. Proschan. *Mathematical theory of reliability*. SIAM, 1996 (cit. on p. 111).

[BDH+19] S. Behnezhad, M. Derakhshan, M. Hajiaghayi, C. Stein, and M. Sudan. “Fully dynamic maximal independent set with polylogarithmic update time”. In: *FOCS*. IEEE. 2019, pp. 382–405 (cit. on p. 51).

[Bel57] R. Bellman. *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 1957 (cit. on p. 49).

[BB10] A. Beloglazov and R. Buyya. “Energy Efficient Allocation of Virtual Machines in Cloud Data Centers”. In: *CCGRID*. IEEE Computer Society, 2010, pp. 577–578 (cit. on p. 50).

[BDJ10a] S. Bereg, A. Dumitrescu, and M. Jiang. “Maximum Area Independent Sets in Disk Intersection Graphs”. In: *Int. J. Comput. Geometry Appl.* 20.2 (2010), pp. 105–118 (cit. on p. 98).

[BDJ10b] S. Bereg, A. Dumitrescu, and M. Jiang. “On Covering Problems of Rado”. In: *Algorithmica* 57.3 (2010), pp. 538–561 (cit. on p. 98).

[BHR19] A. Bernstein, J. Holm, and E. Rotenberg. “Online Bipartite Matching with Amortized $O(\log^2 n)$ Replacements”. In: *J. ACM* 66.5 (2019), 37:1–37:23 (cit. on p. 15).

[BD20] A. Bernstein and A. Dudeja. “Online Matching with Recourse: Random Edge Arrivals”. In: *FSTTCS*. Vol. 182. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 11:1–11:16 (cit. on p. 15).

[BAY+09] A. Berrichi, L. Amodeo, F. Yalaoui, E. Châtelet, and M. Mezghiche. “Bi-objective optimization algorithms for joint production and maintenance scheduling: application to the parallel machine problem”. In: *Journal of Intelligent Manufacturing* 20.4 (2009), pp. 389–400 (cit. on p. 113).

[BGK11] A. Bhalgat, A. Goel, and S. Khanna. “Improved Approximation Results for Stochastic Knapsack Problems”. In: *SODA*. SIAM, 2011, pp. 1647–1665 (cit. on p. 55).

[BHI15] S. Bhattacharya, M. Henzinger, and G. F. Italiano. “Design of Dynamic Algorithms via Primal-Dual Method”. In: *ICALP*. Vol. 9134. Lecture Notes in Computer Science. Springer, 2015, pp. 206–218 (cit. on p. 51).

[BHN19] S. Bhattacharya, M. Henzinger, and D. Nanongkai. “A New Deterministic Algorithm for Dynamic Set Cover”. In: *FOCS*. IEEE Computer Society, 2019, pp. 406–423 (cit. on p. 51).

[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*. SIAM, 2017, pp. 470–489 (cit. on p. 51).

[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*. SIAM, 2019, pp. 1872–1885 (cit. on p. 51).

[BCI+20] S. Bhore, J. Cardinal, J. Iacono, and G. Koumoutsos. “Dynamic Geometric Independent Set”. In: *CoRR* abs/2007.08643 (2020) (cit. on p. 51).

[BBM+17] T. C. Biedl, A. Biniaz, A. Maheshwari, and S. Mehrabi. “Packing Boundary-Anchored Rectangles”. In: *CCCG*. 2017, pp. 138–143 (cit. on p. 98).

[BLK83] J. Blazewicz, J. Lenstra, and A. Kan. “Scheduling subject to resource constraints: classification and complexity”. In: *Discrete Applied Mathematics* 5.1 (1983), pp. 11–24 (cit. on p. 112).

[BKB07] N. Bobroff, A. Kochut, and K. A. Beaty. “Dynamic Placement of Virtual Machines for Managing SLA Violations”. In: *Integrated Network Management*. IEEE, 2007, pp. 119–128 (cit. on p. 50).

[BKK+14] H. Böckenhauer, D. Komm, R. Královic, and P. Rossmanith. “The online knapsack problem: Advice and randomization”. In: *Theor. Comput. Sci.* 527 (2014), pp. 61–72 (cit. on p. 55).

[BP11] N. Boria and V. T. Paschos. “A survey on combinatorial optimization in dynamic environments”. In: *RAIRO - Operations Research* 45.3 (2011), pp. 241–294 (cit. on p. 50).

[BE98] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998, pp. I–XVIII, 1–414 (cit. on p. 114).

[BLS+14] B. Bosek, D. Leniowski, P. Sankowski, and A. Zych. “Online Bipartite Matching in Offline Time”. In: *FOCS*. 2014, pp. 384–393 (cit. on p. 15).

[BFC+20a] T. A. Branca, B. Fornai, V. Colla, M. M. Murri, E. Streppa, and A. J. Schröder. “The Challenge of Digitalization in the Steel Sector”. In: *Metals* 10.2 (2020) (cit. on p. 111).

[BFC+20b] T. A. Branca, B. Fornai, V. Colla, M. M. Murri, E. Streppa, and A. J. Schröder. “The challenge of digitalization in the steel sector”. In: *Metals* 10.2 (2020), p. 288 (cit. on p. 116).

[BKK11] C. Büsing, A. M. C. A. Koster, and M. Kutschka. “Recoverable robust knapsacks: the discrete scenario case”. In: *Optim. Lett.* 5.3 (2011), pp. 379–392 (cit. on p. 55).

[CCC+06] F. T. Chan, S. H. Chung, L. Chan, G. Finke, and M. Tiwari. “Solving distributed FMS scheduling problems subject to maintenance: Genetic algorithms approach”. In: *Robotics and Computer-Integrated Manufacturing* 22.5-6 (2006), pp. 493–504 (cit. on p. 112).

[Cha18] T. M. Chan. “Approximation Schemes for 0-1 Knapsack”. In: *SOSA*. Vol. 61. OASICS. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 5:1–5:12 (cit. on pp. 50, 54, 93).

[CDK+09] K. Chaudhuri, C. Daskalakis, R. Kleinberg, and H. Lin. “Online Bipartite Perfect Matching With Augmentations”. In: *INFOCOM*. 2009, pp. 1044–1052 (cit. on p. 15).

[CZ19] S. Chechik and T. Zhang. “Fully dynamic maximal independent set in expected poly-log update time”. In: *FOCS*. IEEE. 2019, pp. 370–381 (cit. on p. 51).

[CK05] C. Chekuri and S. Khanna. “A Polynomial Time Approximation Scheme for the Multiple Knapsack Problem”. In: *SIAM J. Comput.* 35.3 (2005), pp. 713–728 (cit. on pp. 50, 54, 55, 59).

[CMS07] C. Chekuri, M. Mydlarz, and F. B. Shepherd. “Multicommodity demand flow in a tree and packing integer programs”. In: *ACM Trans. Algorithms* 3.3 (2007), p. 27 (cit. on p. 104).

[CYW08] A. l. Chen, G. Yang, and Z. Wu. “Production scheduling optimization algorithm for the hot rolling processes”. In: *International Journal of Production Research* 46.7 (2008), pp. 1955–1973 (cit. on p. 115).

[CWT14] L. Chen, X. Wang, and L. Tang. “Operation optimization in the hot-rolling production process”. In: *Industrial & Engineering Chemistry Research* 53.28 (2014), pp. 11393–11410 (cit. on p. 115).

[Chr22] N. Christofides. “Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem”. In: *Oper. Res. Forum* 3.1 (2022) (cit. on p. 1).

[CMR20] S. Compton, S. Mitrović, and R. Rubinfeld. “New Partitioning Techniques and Faster Algorithms for Approximate Interval Scheduling”. In: *CoRR* abs/2012.15002 (2020) (cit. on p. 51).

[CJS16] M. Cygan, Ł. Jeż, and J. Sgall. “Online Knapsack Revisited”. In: *Theory Comput. Syst.* 58.1 (2016), pp. 153–190 (cit. on p. 55).

[CMW+19] M. Cygan, M. Mucha, K. Wegrzycki, and M. Wlodarczyk. “On Problems Equivalent to (min, +)-Convolution”. In: *ACM Trans. Algorithms* 15.1 (2019), 14:1–14:25 (cit. on pp. 54, 93).

[DKK+21] C. Damerius, D. Kaaser, P. Kling, and F. Schneider. “On Greedily Packing Anchored Rectangles”. In: *CoRR* abs/2102.08181 (2021) (cit. on pp. 6, 95, 98).

[DKL14] K. Daudjee, S. Kamali, and A. López-Ortiz. “On the online fault-tolerant server consolidation problem”. In: *SPAA*. ACM, 2014, pp. 12–21 (cit. on p. 50).

[DGV08] B. C. Dean, M. X. Goemans, and J. Vondrák. “Approximating the Stochastic Knapsack Problem: The Benefit of Adaptivity”. In: *Math. Oper. Res.* 33.4 (2008), pp. 945–964 (cit. on p. 55).

[Dek95] R. Dekker. “On the use of operations research models for maintenance decision making”. In: *Microelectronics Reliability* 35.9 (1995). Reliability: A Competitive Edge, pp. 1321–1331 (cit. on p. 111).

[DEG+10] C. Demetrescu, D. Eppstein, Z. Galil, and G. F. Italiano. “Dynamic Graph Algorithms”. In: *Algorithms and Theory of Computation Handbook: General Concepts and Techniques*. 2nd ed. Chapman & Hall/CRC, 2010, p. 9 (cit. on p. 50).

[DKM+17] Y. Disser, M. Klimm, N. Megow, and S. Stiller. “Packing a Knapsack of Unknown Capacity”. In: *SIAM J. Discret. Math.* 31.3 (2017), pp. 1477–1497 (cit. on p. 55).

[DP14] R. Duan and S. Pettie. “Linear-Time Approximation for Maximum Weight Matching”. In: *J. ACM* 61.1 (2014), 1:1–1:23 (cit. on p. 16).

[DT15] A. Dumitrescu and C. D. Tóth. “Packing anchored rectangles”. In: *Combinatorica* 35.1 (2015), pp. 39–61 (cit. on pp. 95, 97, 105).

[Ebe20] F. Eberle. “Scheduling and Packing Under Uncertainty”. PhD thesis. University of Bremen, 2020 (cit. on pp. 6, 74, 78, 85).

[EMN+20] F. Eberle, N. Megow, L. Nölke, B. Simon, and A. Wiese. “Fully Dynamic Algorithms for Knapsack Problems with Polylogarithmic Update Time”. In: *CoRR* abs/2007.08415 (2020) (cit. on pp. 74, 78, 85).

[EMN+21] F. Eberle, N. Megow, L. Nölke, B. Simon, and A. Wiese. “Fully Dynamic Algorithms for Knapsack Problems with Polylogarithmic Update Time”. In: *FSTTCS*. Vol. 213. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 18:1–18:17 (cit. on p. 6).

[EBO+19] M. Escudero, J. Balsera, F. Ortega-Fernández, and V. Montequín. “Planning and scheduling with uncertainty in the steel sector: A review”. In: *Applied Sciences* 9.13 (2019), p. 2692 (cit. on p. 114).

[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*. Vol. 107. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 51:1–51:24 (cit. on p. 55).

[FAP98] A. A. Fernandez, R. L. Armacost, and J. J. Pet-Edwards. “Understanding simulation solutions to resource constrained project scheduling problems with stochastic task durations”. In: *Engineering Management Journal* 10.4 (1998), pp. 5–13 (cit. on p. 113).

[FAP96] A. A. Fernandez, R. L. Armacost, and J. J. Pet-Edwards. “The role of the nonanticipativity constraint in commercial software for stochastic project scheduling”. In: *Computers & Industrial Engineering* 31.1-2 (1996), pp. 233–236 (cit. on p. 113).

[FCN+19] X. Fu, F. T. Chan, B. Niu, N. S. Chung, and T. Qu. “A three-level particle swarm optimization with variable neighbourhood search algorithm for the production scheduling problem with mould maintenance”. In: *Swarm and Evolutionary Computation* 50 (2019), p. 100572 (cit. on p. 112).

[GK19] M. Gairing and M. Klimm. “Greedy metric minimum online matchings with random arrivals”. In: *Oper. Res. Lett.* 47.2 (2019), pp. 88–91 (cit. on p. 15).

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability*. Vol. 174. freeman San Francisco, 1979 (cit. on p. 59).

[GL79] G. Gens and E. Levner. “Computational Complexity of Approximation Algorithms for Combinatorial Problems”. In: *MFCS*. Vol. 74. Lecture Notes in Computer Science. Springer, 1979, pp. 292–300 (cit. on p. 54).

[GL80] G. Gens and E. Levner. “Fast approximation algorithms for knapsack type problems”. In: *Optimization Techniques*. Springer, 1980, pp. 185–194 (cit. on p. 54).

[GDA+22] M. Geurtsen, J. B. Didden, J. Adan, Z. Atan, and I. Adan. “Production, maintenance and resource scheduling: A review”. In: *European Journal of Operational Research* (2022) (cit. on p. 112).

[GKK+95] E. Grove, M. Kao, P. Krishnan, and J. Vitter. “Online Perfect Matching and Mobile Computing”. In: *WADS*. Vol. 955. Lecture Notes in Computer Science. 1995, pp. 194–205 (cit. on p. 15).

[GGK16] A. Gu, A. Gupta, and A. Kumar. “The Power of Deferral: Maintaining a Constant-Competitive Steiner Tree Online”. In: *SIAM J. Comput.* 45.1 (2016), pp. 1–28 (cit. on pp. 15, 55).

[GGP+19] A. Gupta, G. Guruganesh, B. Peng, and D. Wajc. “Stochastic Online Metric Matching”. In: *ICALP*. Vol. 132. LIPIcs. 2019, 67:1–67:14 (cit. on p. 15).

[GKS14] A. Gupta, A. Kumar, and C. Stein. “Maintaining Assignments Online: Matching, Scheduling, and Flows”. In: *SODA*. 2014, pp. 468–479 (cit. on p. 15).

[GL12] A. Gupta and K. Lewi. “The Online Metric Matching Problem for Doubling Metrics”. In: *ICALP*. Vol. 7391. Lecture Notes in Computer Science. 2012, pp. 424–435 (cit. on p. 14).

[GKK+17] A. Gupta, R. Krishnaswamy, A. Kumar, and D. Panigrahi. “Online and dynamic algorithms for set cover”. In: *STOC*. ACM, 2017, pp. 537–550 (cit. on p. 51).

[GM20] D. Gupta and C. T. Maravelias. “Framework for studying online production scheduling under endogenous uncertainty”. In: *Comput. Chem. Eng.* 135 (2020), p. 106670 (cit. on p. 114).

[GM19] D. Gupta and C. T. Maravelias. “On the design of online production scheduling algorithms”. In: *Comput. Chem. Eng.* 129 (2019) (cit. on p. 113).

[GKS20] V. Gupta, R. Krishnaswamy, and S. Sandeep. *Permutation Strikes Back: The Power of Recourse in Online Metric Matching*. 2020 (cit. on pp. 14, 46).

[HKM13] X. Han, Y. Kawase, and K. Makino. “Randomized Algorithms for Removable Online Knapsack Problems”. In: *FAW-AAIM*. Vol. 7924. Lecture Notes in Computer Science. Springer, 2013, pp. 60–71 (cit. on p. 55).

[HKM+14] X. Han, Y. Kawase, K. Makino, and H. Guo. “Online removable knapsack problem under convex function”. In: *Theor. Comput. Sci.* 540 (2014), pp. 62–69 (cit. on p. 55).

[HM10] X. Han and K. Makino. “Online removable knapsack with limited cuts”. In: *Theor. Comput. Sci.* 411.44-46 (2010), pp. 3956–3964 (cit. on p. 55).

[HBW+21] V. A. Hauder, A. Beham, S. Wagner, K. F. Doerner, and M. Affenzeller. “Dynamic online optimization in the context of smart manufacturing: an overview”. In: *Procedia Computer Science* 180 (2021), pp. 988–995 (cit. on p. 114).

[Hau21] A. Haupt. “New combinatorial proofs for enumeration problems and random anchored structures”. Doctoral Thesis. Technische Universität Hamburg, 2021 (cit. on p. 98).

[Hen18] M. Henzinger. “The State of the Art in Dynamic Graph Algorithms”. In: *SOFSEM*. Vol. 10706. Lecture Notes in Computer Science. Springer, 2018, pp. 40–44 (cit. on p. 50).

[HNW20] M. Henzinger, S. Neumann, and A. Wiese. “Dynamic Approximate Maximum Independent Set of Intervals, Hypercubes and Hyperrectangles”. In: *SoCG*. Ed. by S. Cabello and D. Z. Chen. Vol. 164. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 51:1–51:14 (cit. on p. 51).

[HK99] M. R. Henzinger and V. King. “Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation”. In: *J. ACM* 46.4 (1999), pp. 502–516 (cit. on p. 50).

[HM21] R. Hoeksma and M. Maat. “A better lower bound for Lower-Left Anchored Rectangle Packing”. In: *CoRR* abs/2102.05747 (2021). arXiv: 2102.05747 (cit. on p. 98).

[HLT01] J. Holm, K. de Lichtenberg, and M. Thorup. “Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity”. In: *J. ACM* 48.4 (2001), pp. 723–760 (cit. on pp. 50, 51).

[IK75] O. H. Ibarra and C. E. Kim. “Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems”. In: *J. ACM* 22.4 (1975), pp. 463–468 (cit. on pp. 50, 54).

[IBM04] IBM Research. *Ponder This | June 2004 Challenge*. https://research.ibm.com/haifa/ponderthis/challenges/June2004.html. 2004 (cit. on p. 97).

[IW91] M. Imase and B. Waxman. “Dynamic Steiner Tree Problem”. In: *SIAM J. Discrete Math.* 4.3 (1991), pp. 369–384 (cit. on pp. 15, 55).

[IL98] Z. Ivkovic and E. L. Lloyd. “Fully Dynamic Algorithms for Bin Packing: Being (Mostly) Myopic Helps”. In: *SIAM J. Comput.* 28.2 (1998), pp. 574–611 (cit. on p. 51).

[IT02] K. Iwama and S. Taketomi. “Removable Online Knapsack Problems”. In: *ICALP*. Vol. 2380. Lecture Notes in Computer Science. Springer, 2002, pp. 293–305 (cit. on p. 55).

[IZ10] K. Iwama and G. Zhang. “Online knapsack with resource augmentation”. In: *Inf. Process. Lett.* 110.22 (2010), pp. 1016–1020 (cit. on p. 55).

[Jan12] K. Jansen. “A Fast Approximation Scheme for the Multiple Knapsack Problem”. In: *SOFSEM*. Vol. 7147. Lecture Notes in Computer Science. Springer, 2012, pp. 313–324 (cit. on pp. 50, 53–55, 68, 79).

[Jan09] K. Jansen. “Parameterized Approximation Scheme for the Multiple Knapsack Problem”. In: *SIAM J. Comput.* 39.4 (2009), pp. 1392–1412 (cit. on pp. 50, 54, 55, 68, 79, 86).

[JK19] K. Jansen and K. Klein. “A Robust AFPTAS for Online Bin Packing with Polynomial Migration”. In: *SIAM J. Discret. Math.* 33.4 (2019), pp. 2062–2091 (cit. on p. 55).

[Jin19] C. Jin. “An Improved FPTAS for 0-1 Knapsack”. In: *ICALP*. Vol. 132. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 76:1–76:14 (cit. on pp. 50, 54, 62, 66, 93).

[KP93] B. Kalyanasundaram and K. Pruhs. “Online Weighted Matching”. In: *J. Algorithms* 14.3 (1993), pp. 478–488 (cit. on pp. 3, 11, 14, 17).

[KVV90] R. Karp, U. Vazirani, and V. Vazirani. “An Optimal Algorithm for On-line Bipartite Matching”. In: *STOC*. 1990, pp. 352–358 (cit. on p. 14).

[Kel99] H. Kellerer. “A Polynomial Time Approximation Scheme for the Multiple Knapsack Problem”. In: *RANDOM-APPROX*. Vol. 1671. Lecture Notes in Computer Science. Springer, 1999, pp. 51–62 (cit. on p. 50).

[KP04] H. Kellerer and U. Pferschy. “Improved Dynamic Programming in Connection with an FPTAS for the Knapsack Problem”. In: *J. Comb. Optim.* 8.1 (2004), pp. 5–11 (cit. on pp. 54, 93).

[KMV94] S. Khuller, S. Mitchell, and V. Vazirani. “On-Line Algorithms for Weighted Bipartite Matching and Stable Marriages”. In: *Theor. Comput. Sci.* 127.2 (1994), pp. 255–267 (cit. on pp. 3, 11, 14, 17).

[KY55] H. W. Kuhn and B. Yaw. “The Hungarian method for the assignment problem”. In: *Naval Res. Logist. Quart* (1955), pp. 83–97 (cit. on pp. 11, 17).

[KS10] A. Kulik and H. Shachnai. “There is no EPTAS for two-dimensional knapsack”. In: *Inf. Process. Lett.* 110.16 (2010), pp. 707–710 (cit. on p. 93).

[KPS17] M. Künnemann, R. Paturi, and S. Schneider. “On the Fine-Grained Complexity of One-Dimensional Dynamic Programming”. In: *ICALP*. Vol. 80. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 21:1–21:15 (cit. on p. 54).

[LOP+15] J. Lacki, J. Oćwieja, M. Pilipczuk, P. Sankowski, and A. Zych. “The Power of Dynamic Distance Oracles: Efficient Dynamic Algorithms for the Steiner Tree”. In: *STOC*. 2015, pp. 11–20 (cit. on p. 15).

[Law79] E. L. Lawler. “Fast Approximation Algorithms for Knapsack Problems”. In: *Math. Oper. Res.* 4.4 (1979), pp. 339–356 (cit. on p. 54).

[LTC14] Y. Li, X. Tang, and W. Cai. “On dynamic bin packing for resource allocation in the cloud”. In: *SPAA*. ACM, 2014, pp. 2–11 (cit. on p. 50).

[Ma18] W. Ma. “Improvements and Generalizations of Stochastic Knapsack and Markovian Bandits Approximation Algorithms”. In: *Math. Oper. Res.* 43.3 (2018), pp. 789–812 (cit. on p. 55).

[MV95] A. Marchetti-Spaccamela and C. Vercellis. “Stochastic on-line knapsack problems”. In: *Math. Program.* 68 (1995), pp. 73–104 (cit. on p. 55).

[MSV19] J. Matuschke, U. Schmidt-Kraepelin, and J. Verschae. “Maintaining Perfect Matchings at Low Cost”. In: *ICALP*. Vol. 132. LIPIcs. 2019, 82:1–82:14 (cit. on p. 15).

[MSV+16] N. Megow, M. Skutella, J. Verschae, and A. Wiese. “The Power of Recourse for Online MST and TSP”. In: *SIAM J. Comput.* 45.3 (2016), pp. 859–880 (cit. on pp. 15, 55).

[MM13] N. Megow and J. Mestre. “Instance-sensitive robustness guarantees for sequencing with unknown packing and covering constraints”. In: *ITCS*. ACM, 2013, pp. 495–504 (cit. on p. 55).

[MN20] N. Megow and L. Nölke. “Online Minimum Cost Matching with Recourse on the Line”. In: *APPROX-RANDOM*. Vol. 176. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 37:1–37:16 (cit. on p. 5).

[MUV06] N. Megow, M. Uetz, and T. Vredeveld. “Models and algorithms for stochastic online scheduling”. In: *Mathematics of Operations Research* 31.3 (2006), pp. 513–525 (cit. on p. 114).

[Meh13] A. Mehta. “Online Matching and Ad Allocation”. In: *Found. Trends Theor. Comput. Sci.* 8.4 (2013), pp. 265–368 (cit. on p. 14).

[MWW06] M. Mika, G. Waligóra, and J. Weglarz. “Modelling Setup Times in Project Scheduling”. In: Springer, 2006. Chap. 6, pp. 131–163 (cit. on p. 113).

[Mon19] M. Monemizadeh. “Dynamic maximal independent set”. In: *CoRR* abs/1906. 09595 (2019) (cit. on p. 51).

[MP89] C. L. Monma and C. N. Potts. “On the Complexity of Scheduling with Batch Setup Times”. In: *Oper. Res.* 37.5 (1989), pp. 798–804 (cit. on pp. 114, 130).

[MWW19] M. Mucha, K. Wegrzycki, and M. Wlodarczyk. “A Subquadratic Approximation Scheme for Partition”. In: *SODA*. SIAM, 2019, pp. 70–88 (cit. on p. 54).

[NRJ+05] R. Nandan, R. Rai, R. Jayakanth, S. Moitra, N. Chakraborti, and A. Mukhopadhyay. “Regulating crown and flatness during hot rolling: A multiobjective optimization study using genetic algorithms”. In: *Materials and Manufacturing Processes* 20.3 (2005), pp. 459–478 (cit. on p. 115).

[NR17] K. Nayyar and S. Raghvendra. “An Input Sensitive Online Algorithm for the Metric Bipartite Matching Problem”. In: *FOCS*. 2017, pp. 505–515 (cit. on pp. 12, 16).

[NSZ06] K. Neumann, C. Schwindt, and J. Zimmermann. “Resource-Constrained Project Scheduling with Time Windows”. In: *Perspectives in Modern Project Scheduling*. Ed. by J. Józefowska and J. Weglarz. International Series in Operations Research & Management Science. Springer, 2006. Chap. 15, pp. 375–407 (cit. on p. 113).

[NML+03] K. Nikolopoulos, K. Metaxiotis, N. Lekatis, and V. Assimakopoulos. “Integrating industrial maintenance strategy into ERP”. In: *Industrial Management & Data Systems* (2003) (cit. on p. 111).

[Oli82] H. J. Olivié. “A New Class of Balanced Search Trees: Half Balanced Binary Search Trees”. In: *RAIRO Theor. Informatics Appl.* 16.1 (1982), pp. 51–71 (cit. on p. 57).

[ÖUH21] A. Özgür, Y. Uygun, and M. Hütt. “A review of planning and scheduling methods for hot rolling mills in steel production”. In: *Comput. Ind. Eng.* 151 (2021), p. 106606 (cit. on p. 115).

[PFA+20] M. Parente, G. Figueira, P. Amorim, and A. Marques. “Production scheduling in the context of Industry 4.0: review and trends”. In: *International Journal of Production Research* 58.17 (2020), pp. 5401–5431 (cit. on p. 113).

[PS21] E. Peserico and M. Scquizzato. “Matching on the Line Admits No $o(\sqrt{\log n})$-Competitive Algorithm”. In: *ICALP*. Vol. 198. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 103:1–103:3 (cit. on pp. 4, 12).

[PRW21] A. Polak, L. Rohwedder, and K. Wegrzycki. “Knapsack and Subset Sum with Small Items”. In: *ICALP*. Vol. 198. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 106:1–106:19 (cit. on p. 93).

[QCT99] X. Qi, T. Chen, and F. Tu. “Scheduling the maintenance on a single machine”. In: *Journal of the operational Research Society* 50.10 (1999), pp. 1071–1078 (cit. on p. 111).

[Rad49] R. Rado. “Some covering theorems (I)”. In: *Proc. London Math. Soc.* 51 (1949), pp. 232–264 (cit. on p. 98).

[Rad51] R. Rado. “Some covering theorems (II)”. In: *Proc. London Math. Soc.* 53 (1951), pp. 243–267 (cit. on p. 98).

[Rad68] R. Rado. “Some covering theorems (III)”. In: *Proc. London Math. Soc.* 42 (1968), pp. 127–130 (cit. on p. 98).

[Rad28] T. Radó. “Sur un problème relatif à un théorème de Vitali”. In: *Fund. Math.* 11 (1928), pp. 228–229 (cit. on p. 98).

[Rag16] S. Raghvendra. “A Robust and Optimal Online Algorithm for Minimum Metric Bipartite Matching”. In: *APPROX-RANDOM*. Vol. 60. LIPIcs. 2016, 18:1–18:16 (cit. on pp. 12, 13, 15–17, 24, 43).

[Rag18] S. Raghvendra. “Optimal Analysis of an Online Algorithm for the Bipartite Matching Problem on a Line”. In: *SoCG*. Vol. 99. LIPIcs. 2018, 67:1–67:14 (cit. on pp. 4, 12–17, 19, 23–25, 28, 45, 46).

[Rhe15] D. Rhee. “Faster fully polynomial approximation schemes for knapsack problems”. Master’s thesis, Massachusetts Institute of Technology. 2015 (cit. on p. 54).

[Ryn08] C. Rynikiewicz. “The climate change challenge and transitions for radical changes in the European steel industry”. In: *Journal of Cleaner Production* 16.7 (2008), pp. 781–789 (cit. on p. 116).

[SSS09] P. Sanders, N. Sivadasan, and M. Skutella. “Online Scheduling with Bounded Migration”. In: *Math. Oper. Res.* 34.2 (2009), pp. 481–498 (cit. on p. 55).

[Sch99] A. Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999 (cit. on p. 104).

[ST03] C. Schwindt and N. Trautmann. “Scheduling the production of rolling ingots: industrial context, model, and solution method”. In: *International Transactions in Operational Research* 10.6 (2003), pp. 547–563. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1475-3995.00427 (cit. on p. 113).

[SC95] J. P. Shewchuk and T. Chang. “Resource-constrained job scheduling with recyclable resources”. In: *European Journal of Operational Research* 81.2 (1995), pp. 364–375 (cit. on p. 113).

[SKL+20] Y. Shin, K. Kim, S. Lee, and H. An. “Online Graph Matching Problems with a Worst-Case Reassignment Budget”. In: *CoRR* abs/2003.05175 (2020) (cit. on p. 15).

[SV16] M. Skutella and J. Verschae. “Robust Polynomial-Time Approximation Schemes for Parallel Machine Scheduling with Job Arrivals and Departures”. In: *Math. Oper. Res.* 41.3 (2016), pp. 991–1021 (cit. on p. 55).

[SEW20] D. Stroud, C. Evans, and M. Weinel. “Innovating for energy efficiency: Digital gamification in the European steel industry”. In: *European Journal of Industrial Relations* 26.4 (2020), pp. 419–437 (cit. on p. 116).

[Tar83] R. E. Tarjan. “Updating a Balanced Search Tree in O(1) Rotations”. In: *Inf. Process. Lett.* 16.5 (1983), pp. 253–257 (cit. on p. 57).

[Tut69] W. T. Tutte, ed. *Recent Progress in Combinatorics: Proceedings of the 3rd Waterloo Conference on Combinatorics*. New York: Academic Press, 1969 (cit. on pp. 6, 95, 96).

[VL81] W. F. de la Vega and G. S. Lueker. “Bin packing can be solved within $1+\varepsilon$ in linear time”. In: *Combinatorica* 1.4 (1981), pp. 349–355 (cit. on pp. 75, 93).

[VGG+20] S. Vögele, M. Grajewski, K. Govorukha, and D. Rübbelke. “Challenges for the European steel industry: Analysis, possible consequences and impacts on sustainable development”. In: *Applied energy* 264 (2020), p. 114633 (cit. on p. 116).

[WL15] S. Wang and M. Liu. “Multi-objective optimization of parallel machine scheduling integrated with multi-resources preventive maintenance planning”. In: *Journal of Manufacturing Systems* 37 (2015), pp. 182–192 (cit. on p. 112).

[Win07] P. Winkler. “Packing Rectangles”. In: *Mathematical Mind-Benders* (2007), pp. 133–134 (cit. on pp. 95, 97).

[WCC12] C. Wong, F. Chan, and S. Chung. “A genetic algorithm approach for production scheduling with mould maintenance consideration”. In: *International Journal of Production Research* 50.20 (2012), pp. 5683–5697. eprint: https://doi.org/10.1080/00207543.2011.613868 (cit. on p. 112).

[WCC14] C. Wong, F. T. Chan, and S. H. Chung. “Decision-making on multi-mould maintenance in production scheduling”. In: *International Journal of Production Research* 52.19 (2014), pp. 5640–5655 (cit. on p. 112).

[Yu96] G. Yu. “On the Max-Min 0-1 Knapsack Problem with Robust Optimization Applications”. In: *Oper. Res.* 44.2 (1996), pp. 407–415 (cit. on p. 55).

[ZSW19] R. Zhang, S. Song, and C. Wu. “Robust scheduling of hot rolling production by local search enhanced ant colony optimization algorithm”. In: *IEEE Transactions on Industrial Informatics* 16.4 (2019), pp. 2809–2819 (cit. on p. 115).

Zusammenfassung

Beim Lösen von Optimierungsproblemen, die sich aus Entscheidungsprozessen in der realen Welt ergeben, ist Unsicherheit ein allgegenwärtiges Phänomen, welches ein enormes Hindernis darstellt. Sei es ein Bäcker, der entscheiden muss, wie viele Backwaren er produziert, ohne zu wissen, was seine Kunden bestellen werden, oder ein Rechenzentrum, in dem die Zuteilung von Rechenressourcen zahlreichen Unsicherheiten unterliegt, von unbekannter zukünftiger Arbeitslast bis hin zu unvorhergesehenen Stromschwankungen; wenn man Optimierungsprobleme in der realen Welt löst, ist es oft unvermeidlich, Entscheidungen zu treffen, ohne ihre vollen Auswirkungen absehen zu können. Die vorliegende Arbeit untersucht, wie mit solchen Unsicherheiten bei der Lösung von Matching- und Packungsproblemen algorithmisch umgegangen werden kann.

Matchingprobleme sind Zuordnungsprobleme, die sich damit befassen, aus einer Menge von Objekten eine Sammlung disjunkter Paare zu berechnen – ein Matching –, welche bezüglich einer bestimmten Zielfunktion optimal ist. Von Packungsproblemen spricht man, wenn Objekte Behältern mit begrenzter Kapazität zugeordnet werden müssen und dabei eine Zielfunktion optimiert werden soll. Beide Problemarten gehören zu den fundamentalen Problemen der kombinatorischen Optimierung und treten in einer Vielzahl von Anwendungen auf, zum Beispiel beim effizienten Packen von Schiffscontainern oder bei der fairen Zuteilung von Spendernieren.

Wir benutzen verschiedene mathematische Modelle, um in der Praxis auftauchende Unsicherheiten abzubilden. Von stochastischen Informationen sprechen wir wenn Problemparameter unbekannt sind aber einer bekannten Wahrscheinlichkeitsverteilung folgen. In Online-Problemen werden Teile der Eingabe nur schrittweise bekannt, wäh-

rend ein (Online-)Algorithmus auf eingehende Informationen sofort, unwiderruflich und ohne Kenntnis der Zukunft reagieren muss. Das Recourse-Setting schwächt diese Unwiderrufbarkeit ab, indem es erlaubt, vergangene Entscheidungen zu ändern, dabei aber die Anzahl der so geänderten Entscheidungen begrenzt. Bei dynamischen Problemen darf die Lösung im Laufe der Zeit sogar beliebig verändert werden, muss jedoch in jedem Update-Schritt schnell neu berechnet werden, das heißt, die Laufzeit sollte polylogarithmisch in der Eingabegröße sein.

Für viele Matching- und Packungsprobleme ist selbst dann, wenn sämtliche Problemparameter bekannt sind, im sogenannten Offline-Setting, das Finden einer optimalen (Offline-)Lösung eine Herausforderung oder sogar unmöglich, zumindest mit derzeit bekannten Methoden. Werden Entscheidungen jedoch unter Unsicherheit getroffen, ohne wichtige Problemparameter zu kennen, so vergrößert sich der Abstand zwischen der Qualität einer auf diese Art (online) erzielbaren Lösung und der einer optimalen Offline-Lösung. Die informationstheoretische Frage, wie genau sich das Fehlen vollständiger Informationen auf die Lösungsqualität auswirkt und wie wir dennoch Algorithmen mit möglichst starken Gütegarantien entwerfen können, ist von zentraler Bedeutung in dieser Arbeit.

Konkret betrachten wir die Berechnung kostenminimaler bipartiter Matchings auf der reellen Linie im Recourse-Setting und liefern hier erste nicht-triviale Resultate. Wir zeigen, dass ein konstantkompetitiver Algorithmus mit einem logarithmischen amortisierten Recoursebudget existiert. Dieser kann angepasst werden, um den Trade-off zwischen kompetitivem Faktor und Recoursebudget zu steuern. Die kompetitive Analyse diese Algorithmus ist bestmöglich. Darüber hinaus entwickeln wir einen einfachen, nahezu optimalen Algorithmus für einen interessanten Spezialfall.

Wir befassen uns auch mit mehreren Versionen des Rucksackproblems im dynamischen Setting. Während dynamische Algorithmen

für Graphenprobleme gut untersucht sind, gibt es kaum Arbeiten zu Packungsproblemen oder zu Problemen, die keine Graphen betreffen, im Allgemeinen. Motiviert durch das theoretische Interesse an Rucksackproblemen und deren praktischer Relevanz wollen wir diese Lücke schließen. Unser Hauptresultat hier ist ein dynamischer Algorithmus für eine beliebige Anzahl von Rucksäcken, welcher $(1-\varepsilon)$-approximative Lösungen mit einer Update-Zeit von etwa $(\frac{1}{\varepsilon} \cdot \log n)^{O(\frac{1}{\varepsilon})}$ berechnet, wobei n die derzeitige Anzahl zu packender Objekte ist. Wir begründen die superpolynomielle Abhängigkeit von ε unter der Annahme $\mathcal{P} \neq \mathcal{NP}$ und beschreiben weitere, noch effizientere Algorithmen für relevante Spezialfälle.

Als weiteres Packungsproblem betrachten wir das Packen verankerter Rechtecke. Gegeben ist hier eine Menge von Punkten P im Einheitsquadrat $[0, 1]^2$, und wir suchen eine Menge $\mathcal{S}$ von achsenparallelen, innendisjunkten Rechtecken, die im Einheitsquadrat liegen und jeweils an einem Punkt $p \in P$ verankert sind. Dabei wollen wir die von $\mathcal{S}$ überdeckte Fläche maximieren. Bei der prominentesten Variante muss der Anker p in der linken unteren Ecke des Rechtecks liegen. Hier entwickeln wir Algorithmen für zwei Formen von Ressourcenerhöhung, welche jeweils die Bedingung des Verankerns und der Disjunktheit aufweichen, und zeigen, dass die Algorithmen eine optimale Lösung ohne Ressourcenerhöhung mit dem Faktor 1 beziehungsweise $(1 - \varepsilon)$ approximieren. Für das Problem, in dem Rechtecke in ihrer Mitte verankert sind, präsentieren wir ein Polynomialzeit-Approximationsschema.

Zuletzt behandeln wir ein praktisches Problem, das durch ein Projekt im Warmwalzwerk eines multinationalen Stahlherstellers motiviert ist und die Allokation und Wartung wiederverwendbarer Ressourcen in ressourcenbeschränkten Produktionsumgebungen betrifft. Die Aufgabe ist dreigeteilt: Ressourcen müssen Produktionsjobs zugeordnet werden, um deren Ressourcenbedarf zu decken; die Wartung der Ressourcen – gemäß der obigen Zuordnung – muss auf heterogenen

Wartungsmaschinen mit ressourcentypabhängigen Rüstzeiten geplant werden; und die Durchführung der ressourcenbeschränkten Produktionsjobs muss geplant werden. Dabei sollen sowohl Wartezeiten, die von der Ressourcen-zuweisung abhängen, als auch Wartekosten, die durch langsame Wartung verursacht werden können, minimiert werden. Wir nennen dies das SAMS-Problem (Simultaneous Allocation and Maintenance Scheduling). Eine zusätzliche Schwierigkeit liegt in der Online-Natur des Produktionsablaufs, in dem immer nur die nächsten k Produktionsjobs bekannt sind; wir nennen dies Lookahead k.

Wir stellen mehrere ganzzahlige lineare Programme (ILP) für das SAMS-Problem vor, die im Offline-Setting optimale Lösungen berechnen, und zeigen, wie diese ILPs im Online-Setting mit Lookahead k genutzt werden können. Darüber hinaus entwickeln wir eine effiziente Heuristik, die das SAMS-Problem in drei Phasen unterteilt und Matching- und Scheduling-Techniken verwendet, um Lösungen für SAMS mit Lookahead k zu erhalten. Die Heuristik wird anhand von Datensätzen aus der Praxis und den entwickelten ganzzahligen linearen Programmen bewertet und erzielt gute Resultate, insbesondere im Bezug auf die Minimierung der Wartezeit, welche das wichtigere der beiden Optimierungsziele darstellt.