A Tale of Two Packing Problems:

Improved Algorithms and Tighter Bounds for Online Bin Packing and the Geometric Knapsack Problem

A dissertation submitted towards the degree of

Doctor of Natural Science

of the Faculty of Mathematics and Computer Science
of Saarland University

by
Sandy Heydrich

Saarbrücken
– 2018 –

Tag des Kolloquiums: 12.06.2018

Dekan der Fakultät für Mathematik und Informatik:
Prof. Dr. Sebastian Hack

Prüfungsausschuss:
Prof. Dr. Markus Bläser (Vorsitzender)
Prof. Dr. Rob van Stee
Prof. Dr. Dr. h.c. mult. Kurt Mehlhorn
Prof. Dr. Fabrizio Grandoni
Dr. Andreas Wiese
Dr. Antonios Antoniadis (Akademischer Mitarbeiter)

Abstract

In this thesis, we deal with two packing problems: the online bin packing and the geometric knapsack problem. In online bin packing, the aim is to pack a given number of items of different size into a minimal number of containers. The items need to be packed one by one without knowing future items. For online bin packing in one dimension, we present a new family of algorithms that constitutes the first improvement over the previously best algorithm in almost 15 years. While the algorithmic ideas are intuitive, an elaborate analysis is required to prove its competitive ratio. We also give a lower bound for the competitive ratio of this family of algorithms. For online bin packing in higher dimensions, we discuss lower bounds for the competitive ratio and show that the ideas from the one-dimensional case cannot be easily transferred to obtain better two-dimensional algorithms.

In the geometric knapsack problem, one aims to pack a maximum weight subset of given rectangles into one square container. For this problem, we consider offline approximation algorithms. For geometric knapsack with square items, we improve the running time of the best known PTAS and obtain an EPTAS. This shows that large running times caused by some standard techniques for geometric packing problems are not always necessary and can be improved. Finally, we show how to use resource augmentation to compute optimal solutions in EPTAS-time, thereby improving upon the known PTAS for this case.

Zusammenfassung

In dieser Arbeit betrachten wir zwei Packungsprobleme: Online Bin Packing und das geometrische Rucksackproblem. Bei Online Bin Packing versucht man, eine gegebene Menge an Objekten verschiedener Größe in die kleinstmögliche Anzahl an Behältern zu packen. Die Objekte müssen eins nach dem anderen gepackt werden, ohne zukünftige Objekte zu kennen. Für eindimensionales Online Bin Packing beschreiben wir einen neuen Algorithmus, der die erste Verbesserung gegenüber dem bisher besten Algorithmus seit fast 15 Jahren darstellt. Während die algorithmischen Ideen intuitiv sind, ist eine ausgefeilte Analyse notwendig um das Kompetitivitätsverhältnis zu beweisen. Für Online Bin Packing in mehreren Dimensionen geben wir untere Schranken für das Kompetitivitätsverhältnis an und zeigen, dass die Ideen aus dem eindimensionalen Fall nicht direkt zu einer Verbesserung führen.

Beim geometrischen Rucksackproblem ist es das Ziel, eine größtmögliche Teilmenge gegebener Rechtecke in einen einzelnen quadratischen Behälter zu packen. Für dieses Problem betrachten wir Approximationsalgorithmen. Für das Problem mit quadratischen Objekten verbessern wir die Laufzeit des bekannten PTAS zu einem EPTAS. Die langen Laufzeiten vieler Standardtechniken für geometrische Probleme können also vermieden werden. Schließlich zeigen wir, wie Ressourcenvergrößerung genutzt werden kann, um eine optimale Lösung in EPTAS-Zeit zu berechnen, was das bisherige PTAS verbessert.

Acknowledgments

> People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.
>
> Donald E. Knuth

I deeply thank my advisor Rob van Stee for introducing me to the intriguing world of bin packing, for being an incredibly reliable and pleasant person to work with, and an inexhaustible source of optimism and good ideas.

I want to thank Andreas Wiese for always being available for discussions and questions (even at 8 am in the morning!), for patiently introducing me to geometric packing problems, and for being a lot of fun to work with.

I am grateful to Kurt Mehlhorn for his support and for making the algorithms group at MPI what it is: an incredible place to work. Thanks to Kurt's initiative and the support by all three of you, I was awarded a Google PhD Fellowship, and I am very thankful for that.

My sincere thanks also go to Fabrizio Grandoni for agreeing to review this thesis.

I also want to thank all the people at MPI. Special thanks go to Stephan Friedrichs, Kevin Schewior, and Andreas Schmid; it was a lot of fun to work with you and share the curiosities of a PhD student's everyday life with you.

I thank my good friends Martin Bromberger, Andrea Fischer, Ralf Jung, Jan-Oliver Kaiser, Jana Rehse, and Tim Ruffing. We had a whole lot of fun and your friendship carried me through the rougher times of this PhD journey. I will miss the lunch breaks on Thursdays terribly. Special thanks go to Andrea, Jan-Oliver, Jana, and Tim, who proofread this thesis. Andrea, you especially have put great effort in helping me to polish the presentation of this thesis, and I am very thankful for our intense discussions and your valuable, well-founded advice.

Ich danke meinen Eltern Sylke und Jörg, dass sie immer hinter mir standen und stehen und mich bedingungslos unterstützen. Ohne euch wäre diese Arbeit nicht möglich gewesen und ohne euch wäre ich nicht der Mensch, der ich bin.

I saved the best for last: I thank my husband Elias for his unconditioned and unwavering love and support. I wouldn't have come so far without you.

Contents

1 Introduction

In this thesis, we study *packing problems*. Imagine you are moving to a new apartment and want to pack all of your belongings into boxes for transportation. Of course, you want to make as few trips as possible between your new and old home, so your goal is to use as few boxes as possible in order to make efficient use of the space available in the truck. For another example, imagine you are the publisher of a newspaper and you receive offers from companies who want to advertise their services and products in your newspaper. Each potential ad has a certain length and width and the companies are willing to pay different prices for their ads to be printed. Your goal is to select some of these ads and arrange them on one newspaper page such that you maximize the revenue you get.

It is of course easy to think of a multitude of similar problems: optimizing how many transporters a logistics company has to send in order to fulfill their contracts, minimizing the number of storage devices required in order to store a set of files (where a single file cannot be distributed over multiple devices), or selecting a set of requests you want to execute on a server, where each request requires a certain amount of memory and yields in turn a certain profit.

When considering such problems, we notice that they are all *optimization problems*: Instead of only looking for *some* solution to a given problem instance, we are also given a measure of quality for any possible solution. Our goal is to optimize this solution quality. In our examples, this measure is the number of moving boxes we need for all our belongings, the total profit of the selected ads, the number of hard drives to store all data, and so forth. Optimization problems are a fundamental notion in algorithms research. In an enormous number of applications we require a good solution instead of merely any solution. For example, you would usually want your navigation device to give you the *shortest* route from one place to another.

Amongst optimization problems in general, one might find that packing problems – such as the ones described in the beginning – play a particularly important role as they pose a very fundamental challenge: How can we use limited space most efficiently? We encounter this question not only in concrete practical applications but also often as a subproblem of other, more complex problems. Results on packing problems can therefore give fundamental insights into many other problems from computer science. This potential impact on the broader computer science community makes packing problems particularly interesting.

In order to find meaningful answers for such questions in realistic settings, we often have to consider additional factors. For example, we might be missing some information, or our computing power may be limited. Three such additional factors influencing the design and quality of algorithms which are relevant in this thesis are the following:

Incomplete information Let us consider the example of moving house. You will probably not have a complete inventory of your belongings that lists each single item together with its exact size. You usually also do not want to take inventory of them as you probably do not have the space and time to take out all items at once and catalogue them. What you will probably do instead is take out the items one by one, pack them into the boxes as they come, and try to avoid repacking items again and again. Similar restrictions are reasonable for many other applications of such packing problem, thus motivating the so-called *online model*: The items arrive over time and whenever an item arrives, we immediately have to decide where to pack this item, without knowing which items will arrive in the future and without being able to revoke our packing decisions later on.

While we see that the online model is well-motivated from practical applications, it is also noteworthy that it poses an interesting theoretical question, namely examining how well we can solve problems if we do not have full information about the input. This, in effect, can also help in identifying why a problem is hard: Online algorithms can make clear why and in which way decisions need to be global in order to achieve good results by showing the impact of the impossibility of global decisions.

Restricted computational resources Many packing as well as other optimization problems arising in applications are NP-complete and thus solving these problems exactly might not be feasible. Instead, we need to find approximate solutions that provide a good guarantee on the solution's quality but can be computed quickly, within the timeframe we are able to allocate to this problem. When considering the newspaper ad example, we might not be willing to invest a large amount of time in finding the optimal selection of ads and their arrangement. Instead, we might be satisfied with finding a solution that is guaranteed to yield, e.g., at least 99% of the profit of the optimal solution. This motivates research on *approximation algorithms*.

It is easy to see that this can be very interesting for practical applications, but once again, quantifying this kind of tradeoff between running time and solution quality is also an interesting theoretical question in itself. Approximation algorithms can for example illustrate which structural information about a problem can be leveraged in polynomial time – thus pointing to which parts of the structure make the problem easier.

Soft problem constraints When designing algorithms, we usually deal with two different types of constraints: Inherent problem constraints, such as the capacity of our container, and algorithmic constraints imposed by the model, such as using only polynomial running time. In the two situations described above we impose additional algorithmic constraints – the lack of complete information and the requirement of polynomial running time. But we might also be in the opposite situation where we can slightly *relax* inherent constraints of the problem itself.

Imagine that when packing your belongings for moving house, you might end up with one box with some free space left and a single item which is just slightly too large to be packed into this box. If we assume for a moment that the objects in this box are not too fragile, you will probably be very happy with squeezing this additional item into the already partly-filled box instead of using an additional one. This illustrates that in some situations, you might be willing to relax some of the constraints that arise from the problem itself – like the capacity of the container – in order to obtain a better solution. This concept is captured in the notion of *resource*

augmentation: We might be allowed to make our container a bit larger in order to improve the solution quality.

And while, again, this is well-motivated from practical examples, we also pose a more fundamental, theoretical question here: If a problem is very hard to solve, what can we gain when relaxing some of the constraints, and to which extent do we need to relax them? This can be another building block in identifying what makes a hard problem hard.

The real world-inspired packing problems described in the beginning can all be modeled by two computational problems: *bin packing* and *the knapsack problem*. They are the focus of this thesis.

In both problems, we are asked to pack items into containers, and in both problems each item has a particular size and each container has a particular capacity. In bin packing, we are given an unbounded number of containers, and we want to pack *all items* into as few of them as possible. In the knapsack problem, on the other hand, we only have a *single container* and want to select and pack a subset of the items into it. Here, each item also has an associated profit and our goal is to maximize the total profit of those items that we pack. When you are packing your belongings for moving house, you are solving a bin packing problem. When trying to select and arrange newspaper ads, you are solving a knapsack problem.

Note that both problems can of course be considered in one or in multiple dimensions. In the one-dimensional version, items and containers have a scalar size, which is for example the case if you want to store files on as few as possible storage devices. In two dimensions, items and containers become rectangles, in three dimensions they become boxes, and so forth.[1] While even the one-dimensional variants are NP-hard, the problems become even harder in higher dimensions because we have to fulfill multiple geometric – and thus intertwined – constraints.

Both problems have a long history in computer science. The bin packing problem has been studied already since the 1950s [35, 58] and has not lost relevance since then. In fact, it could be said that bin packing spawned two independent research areas: online algorithms and approximation algorithms [98]. Many techniques and ideas later applied to other problems in these fields were first used in bin packing. This is another reason for the continually high interest in this old problem besides its broad applicability.

The knapsack problem is no less fundamental and practically relevant than bin packing. They are also closely related: Knapsack is a natural dual of bin packing, and in fact often appears as a subproblem in bin packing problems in form of a separation oracle. First results about the universality of the knapsack LP were published more than one hundred years ago [92] and since the 1950s the problem received growing interest [34].

The fundamentality of the bin packing and knapsack problem motivates ongoing research on these long-standing but still not completely understood problems. In addition, the online model, approximation algorithms, and resource augmentation are highly relevant in practical examples and pose interesting theoretical questions. With this thesis, we want to contribute some new ideas and insights to this field of research.

[1] What we describe here is in fact two-dimensional *geometric* bin packing and knapsack. We will describe the difference between the geometric and non-geometric versions in chapters 3 and 6

1.1 Contributions and organization of this thesis

After this introduction, we will give some general definitions and notations used in this thesis in chapter 2. We can then start describing the examined problems and our results.

Part I contains our results about online bin packing.

- We start by giving the formal definitions needed for the bin packing problem and by discussing related work in **chapter 3**.
- In **chapter 4**, we present a new family of algorithms for the online one-dimensional bin packing problem. We first discuss the previously best family of algorithms for this problem, Super Harmonic, in particular pointing out its weaknesses and limitations. We then discuss the ideas we developed in order to overcome these weaknesses, leading to a new and better family of algorithms. In particular, our result breaks a lower bound applicable to the previously known algorithms and reduces the gap to the general lower bound by more than 15%. We furthermore show a new lower bound for our family of algorithms. In a second step, we show how an additional and rather small change can improve the performance of our framework even further, reducing the gap by more than 20% compared to Super Harmonic.
- In **chapter 5**, we discuss lower bounds for multi-dimensional geometric online bin packing. We show lower bounds for general algorithms but also for different classes of algorithms that are motivated by the literature and our algorithm from chapter 4.

Part II is concerned with the offline geometric knapsack problem.

- **Chapter 6** introduces the required formal definitions and discusses prior work.
- In **chapter 7**, we discuss the special case of geometric knapsack where all input items are squares rather than arbitrary rectangles. While a so-called polynomial time approximation scheme, that is an approximation scheme that can approximate the optimal solution with arbitrary precision, was already known, this algorithm has a rather large (while still polynomial) running time. We show how this approximation scheme can be adjusted to allow for a much smaller running time while maintaining the approximation guarantee.
- Finally, in **chapter 8**, we discuss the general geometric knapsack problem, i.e., allowing arbitrary rectangles as input items, in the setting where resource augmentation is allowed. We show that we can find a solution that is at least as good as the optimal one and also improves the running time of the best known approximation scheme that existed before (which did *not* find an optimal but only an approximate solution).

1.2 Publications

The results presented in this thesis were published in the following articles.

[65] Sandy Heydrich and Rob van Stee. "Beating the Harmonic Lower Bound for Online Bin Packing." In: *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*. Ed. by Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi. Vol 55. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 41:1-41:14. ISBN: 978-3-95977-013-2. DOI: 10.4230/LIPIcs.ICALP.2016.41.

[18] David Blitz, Sandy Heydrich, Rob van Stee, André van Vliet, and Gerhard J. Woeginger. "Improved Lower Bounds for Online Hypercube and Rectangle Packing." In: CoRR abs/1607.01229 (2016). URL: https://arxiv.org/abs/1607.01229.

[66] Sandy Heydrich and Andreas Wiese. "Faster approximation schemes for the two-dimensional knapsack problem." In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16–19*. Ed. by Philip N. Klein. SIAM, 2017, pp. 79-98. ISBN: 978-1-61197-478-2. DOI: 10.1137/1.9781611974782.6.

A preliminary version of the results of chapter 4 was published in ICALP 2016 [65]. Compared to that publication, the presentation of the results was improved and extended for this thesis, and minor issues have been corrected.

The results of chapter 5 are available online [18] and currently under review for publication. The results described in section 5.2 and many concepts and ideas used in section 5.1 were derived by David Blitz in his Master's thesis [17]. We improve his lower bound for square packing very slightly from 1.68025 to 1.68078 and present for the first time a full and complete proof of all the steps needed to obtain these results. For rectangle packing, we now only claim a 1.859 lower bound, as we were unable to give a formal proof of all the details needed to obtain Blitz's claimed bound of 1.907.

The results of chapter 7 and chapter 8 were published in SODA 2017 [66]. For this thesis, the presentation has been improved.

2 Definitions and Notations

In the following, we introduce some important models and notions that are used in this thesis to handle settings with restricted resources or incomplete information.

2.1 Approximations

Many optimization problems that are both practically and theoretically relevant cannot be solved optimally in polynomial time unless P = NP. In that case, we might ask how well (w.r.t. the quality measure) we can approximate the optimal solution in polynomial time. This leads to the notion of approximation algorithms and approximation ratios defined below.

Definition 1 (Approximation ratio). *Let Π be an optimization problem, $\mathcal{I}$ the set of all input instances of Π and $\mathcal{A}$ an algorithm for Π that runs in polynomial time. Denote by $\mathrm{OPT}(I)$ the value of the optimal solution for instance $I \in \mathcal{I}$ and by $\mathcal{A}(I)$ the value of the solution of $\mathcal{A}$ for I. We say that $\mathcal{A}$ has an* approximation ratio of α *(or $\mathcal{A}$ is an α-approximation algorithm) if, for any input instance $I \in \mathcal{I}$,*

- $\mathcal{A}(I) \leq \alpha \mathrm{OPT}(I)$, *in case Π is a minimization problem, or*
- $\mathrm{OPT}(I) \leq \alpha \mathcal{A}(I)$, *in case Π is a maximization problem.*

Note that this definition implies that the approximation ratio is always at least 1. An algorithm with approximation ratio of exactly 1 solves the problem optimally for every input instance in polynomial time.

In some cases, one can approximate a problem within a constant factor, and in the best case, one even knows a family of algorithms that can find an approximation algorithm with *any* desired constant approximation ratio. This concept is formalized in the notion of a *polynomial time approximation scheme*.

Definition 2 (PTAS). *A family of algorithms is called a* polynomial time approximation scheme (PTAS) *if, given a constant parameter $\varepsilon > 0$, it finds a $(1+\varepsilon)$-approximation in a running time that is polynomial in the input size n.*

Note that the running time of a PTAS needs to be polynomial for *constant* ε, i.e., it will usually depend on ε. This dependence on ε might be arbitrarily bad, e.g., one can get running times of the form $n^{2^{\varepsilon}}$ or even worse functions of ε in the exponent of n. One could therefore try to find a PTAS with a better guarantee on the running time w.r.t. ε, which leads us to the definition of *efficient polynomial time approximation schemes* given next.

Definition 3 (EPTAS). *A family of algorithms is called an* efficient polynomial time approximation scheme (EPTAS) *if it is a* PTAS *with running time of the form $f(1/\varepsilon) \cdot n^{O(1)}$ for some function f.*

Note that this means that the exponent of n does not depend on ε, only the constant factor $f(1/\varepsilon)$ does. In order to be able to handle dependencies on ε more clearly, we introduce the following notation: Whenever we mean a constant that depends on ε, we write $O_\varepsilon(1)$, for a constant that does not depend on ε we write $O(1)$. Using this notation, an EPTAS is a PTAS with running time $O_\varepsilon(1) \cdot n^{O(1)}$.

However, the constant $f(1/\varepsilon)$ might still be exponential in $1/\varepsilon$, so we might even want to go further.

Definition 4 (FPTAS). *A family of algorithms is called a* fully polynomial time approximation scheme (FPTAS) *if it is a* PTAS *with running time polynomial in n and in $1/\varepsilon$.*

Note that no FPTAS can exist for strongly NP-hard problems unless P = NP [105].

Another concept appearing in this thesis is that of *asymptotic* analysis. In some settings and problems, there are very small input instances known that obstruct a good approximation ratio, but no large problematic inputs are known. In this case, it can be more insightful to find out whether an algorithm can do better on large inputs than on small, maybe pathological instances. This is for example the case for bin packing (see also chapter 3). In such cases, it makes sense to consider the *asymptotic approximation ratio* instead of the *absolute* approximation ratio defined above.

Definition 5 (Asymptotic approximation ratio). *An algorithm $\mathcal{A}$ for problem Π with set of input instances $\mathcal{I}$ has* asymptotic approximation ratio α *if there is a constant c such that for any input instance $I \in \mathcal{I}$ we have*

- $\mathcal{A}(I) \leq \alpha \mathrm{OPT}(I) + c$, *in case Π is a minimization problem, or*
- $\mathrm{OPT}(I) \leq \alpha \mathcal{A}(I) + c$, *in case Π is a maximization problem.*

Similarly, we define an *asymptotic* polynomial time approximation scheme as a polynomial-time algorithm with asymptotic approximation ratio $1+\varepsilon$ for any constant $\varepsilon > 0$ (an APTAS); the same goes for an asymptotic EPTAS and asymptotic FPTAS (AFPTAS).

2.2 Online problems

We now introduce a second interesting model for algorithmic optimization problems: the online model. Recall the example from before about packing boxes for moving house: The items are taken out of shelves and cupboards one by one and we want to pack each item into a box immediately. To model such a situation formally, we assume the following: The input of the problem is not given as a whole to the algorithm beforehand, but is presented piece by piece. Furthermore, the algorithm has to make decisions during the revelation of the input, i.e., at a point in time where the whole input is not yet known. These decisions usually affect the parts of the input that have already been revealed and cannot be revoked later on when new information is gained. Note that the algorithm also does not know the length of the input sequence. To stay with our example, whenever a new task arrives, we immediately have to assign it to a worker (or hire a new worker), who will execute this task. We cannot reassign tasks once they have been scheduled and only when the current task is assigned to a worker, we will gain knowledge of the next task.

Online problems pose the question of how well we can solve a problem under such a model of incomplete information. We want to quantify the impact of incomplete information on the computability of optimal solutions, asking how much – in terms of solution quality – we lose due to the lack of information, compared to the situation where we have all information upfront. This is somewhat orthogonal to the notion of approximation algorithms, and therefore we are usually not restricting the allowed running time. The algorithm is indeed often assumed to have unbounded computational power and allowed to use unbounded (while of course finite) running time and space.

In order to quantify the loss in solution quality, we use the notion of *competitive analysis*, comparing the quality of the solution of the online algorithm to the quality of the best possible solution (also often referred to as optimal offline solution) that can be found if one knows the whole input from the beginning.[1] For this, we use the so-called competitive ratio introduced by Sleator and Tarjan [102].

Definition 6 (Competitive ratio)**.** *Let Π be an optimization problem, $\mathcal{I}$ the set of input instances of Π and $\mathcal{A}$ an online algorithm for Π. Denote by $\mathrm{OPT}(I)$ the quality of the optimal offline solution for instance $I \in \mathcal{I}$ and by $\mathcal{A}(I)$ the quality of the solution of $\mathcal{A}$ for I. $\mathcal{A}$ has* competitive ratio *α (or is α-competitive) if for every input instance $I \in \mathcal{I}$ we have*

- $\mathcal{A}(I) \leq \alpha \mathrm{OPT}(I)$*, in case Π is a minimization problem, or*
- $\mathrm{OPT}(I) \leq \alpha \mathcal{A}(I)$*, in case Π is a maximization problem.*

Again, the competitive ratio is defined in such a way that it is always at least 1. A 1-competitive algorithm solves a problem optimally for all input instances.

As with approximations, in some cases we are interested in the behavior of the algorithm on very large inputs only. We therefore also define the notion of *asymptotic* competitive ratio.

Definition 7 (Asymptotic competitive ratio)**.** *Let Π be an optimization problem, $\mathcal{I}$ the set of input instances of it and $\mathcal{A}$ an online algorithm for Π. Denote by $\mathrm{OPT}(I)$ the quality of the optimal offline solution for instance $I \in \mathcal{I}$ and by $\mathcal{A}(I)$ the quality of the solution of $\mathcal{A}$ for I. $\mathcal{A}$ has* asymptotic competitive ratio *α (or is asymptotically α-competitive) if there is some constant c such that for every instance $I \in \mathcal{I}$ we have*

- $\mathcal{A}(I) \leq \alpha \mathrm{OPT}(I) + c$*, in case Π is a minimization problem, or*
- $\mathrm{OPT}(I) \leq \alpha \mathcal{A}(I) + c$*, in case Π is a maximization problem.*

When working with online models, it is often helpful to think of an adversary that defines the input, and who is moreover adaptive. Whenever the online algorithm makes a decision, the adversary decides on the worst possible continuation of the current input sequence. The worst possible input is the one that maximizes the competitive ratio, i.e. the gap between the algorithm's solution and the optimal offline solution. The adversary can also decide to stop the input at any time, if this is disadvantageous for the algorithm.

[1] This does *not* mean that we compare to solutions computable by an offline polynomial-time *algorithm*; rather think of the best solution an oracle could provide for the problem.

Part I

Online Bin Packing

3 An Overview of Bin Packing

Informally, bin packing can be seen as the following problem: Given a set of items of different sizes, pack them into as few as possible bins of unit size. As detailed in chapter 1, this is a very fundamental and broadly applicable problem, with a long list of results and algorithms. Some bin packing problems are of such great importance that they are also studied in the literature as separate problems.

One of them is the *cutting stock problem*. Imagine the following situation: A merchant is offering metal rods and has a supply of rods of a fixed length L. Customers request pieces of certain lengths $\leq L$. The merchant wants to cut all requested pieces from as few as possible length L rods in order to minimize the amount of wasted material. This is a simple instance of a one-dimensional bin packing problem. For an example in higher dimension, imagine that the merchant is selling cutouts of sheet metal, and is given a list of rectangular pieces with certain side lengths that need to be cut out of fixed size raw material sheets. Also the online model is very relevant in this example: Whenever a customer arrives, the merchant should cut the requested piece immediately, without waiting for a large number of customers to arrive first before fulfilling all their requests at once even if this might reduce the amount of material wasted. The cutting stock problem was studied since the 1930s [79, 78, 35, 58] and a large number of professional software solutions for industry applications is available (see for example [2, 96, 103]).

On top of that, bin packing has a strong connection to scheduling problems (see e.g. [57, 27, 29]), a large area of research in itself. For illustration, assume we have a set of jobs that need to be done, where each job takes a certain amount of time to finish. We want to find the minimal number of workers we need to hire and assign jobs to these workers such that no worker has to work longer than a specific amount of time a day as imposed by labour law regulations.

Next, we describe the problem formally in the offline and online setting as well as in the one- and multi-dimensional version.

3.1 Problem definition

One-dimensional bin packing In the bin packing problem, we are given a set of n items $1, \dots, n$, each with an associated size $s_i \in [0, 1]$, and an unlimited supply of bins of size 1. The goal is to pack all the items into bins, i.e., assign each item to a bin such that the total size of the items in any bin is at most one, and the number of used bins (i.e., bins that contain at least one item) is minimized.

In the online version of the bin packing problem, the input items arrive one by one over time, and whenever an item arrives, the algorithm immediately has to decide where to pack this item. It does not know which and how many items arrive in the future and it cannot revoke its packing decision later on.

Multi-dimensional bin packing A natural generalization of bin packing to higher dimensions asks the following: Given bins of capacity one in all d dimensions and a set of items $1,\dots,n$, each item i with d sizes s_i^j for $j = 1,\dots,d$, pack the items into as few as possible bins while making sure that for each set of items I_B packed in one bin B, $\sum_{i\in I_B} s_i^j \leq 1$. This is also called the *vector bin packing problem*. In such a setting, we have to respect several capacity constraints, which are however independent of each other; e.g. think of packing items that must not exceed size and weight constraints.

Another obvious generalization is the multi-dimensional *geometric* bin packing problem, which resembles the packing of objects with multiple spatial dimensions, such as packing three-dimensional boxes. Each item i is now a d-dimensional box of side length $s_i^j \in [0,1]$ in dimension j. Similarly, the bins are unit-size hypercubes. In order to solve the problem, we now not only have to assign items to bins, but we have to assign each item a specific position within this bin, in such a way that items are axis-aligned, no two items overlap and no item exceeds the bin's boundary. Thus, the different constraints in this problem are highly related.

Note that the vector and the geometric multi-dimensional bin packing problems are rather different; while for vector bin packing, it is easy to compute whether a given set of items fits into a single bin (just add up their sizes in every dimension), this is a hard problem in the geometric setting, where we have to try out different positionings to see if the items fit. This thesis only deals with geometric bin packing.

In geometric bin packing, one can also consider more specialized models. For example, we can either allow to rotate the items[1] or forbid this, forcing items to be packed in the orientation specified in the input[2].

Another interesting setting is the case when items are not arbitrary boxes, but hypercubes instead. Then, each item is again characterized by a scalar size which represents the side length in all dimensions. That way, it becomes simpler to adapt one-dimensional algorithms – which can only handle a scalar size for an item – for the two-dimensional case, allowing us to examine how such algorithms behave in higher dimensions.

Of course, multi-dimensional bin packing problems can again be considered in an offline or online setting analogously to the one-dimensional case.

3.2 Related work

3.2.1 One-dimensional bin packing

Offline version The bin packing problem has been present in the literature since the 1950s [35, 58]. It was one of the very first problems that was considered in the context of approximations, and one of the first problems for which formal proofs for a certain quality *guarantee* for the solution of an approximation algorithm were given [55]. Garey et al. [56] formulated it as a generalization of various memory allocation problems and it is well known that the problem is NP-hard [54]. It is even strongly NP-complete and so no FPTAS can exist for this problem [53]. From the NP-hardness proof one can immediately conclude that no algorithm can approximate the problem with an absolute ratio better than 3/2 unless P = NP, as the NP-hardness proof shows that an algorithm that could distinguish between bin packing instances

[1] Usually, we consider rotations by 90 degrees, as the problem becomes even more complex as soon as items are not axis-parallel anymore.

[2] This might be necessary when transporting sensitive cargo, such as refrigerators.

which can be packed in 2 or 3 bins can be used to solve the 2-Partition problem. This barrier is the reason why *asymptotic* analysis was always considered to be the most interesting model for this problem.

One of the simplest algorithms for the bin packing problem is NextFit: Pack the first item in the first bin and call this the "current" bin, and then, for each item, pack it into the current bin if it fits, otherwise pack it into a new bin which then becomes the new current bin. It is very easy to see that this algorithm is an absolute 2-approximation (any two consecutive bins – other than the last one – must contain items of total size more than 1), and it can be improved to an asymptotic 1.691-approximation when considering the items in decreasing order of size [5]. Some similar algorithms that were proposed later further improved the approximation guarantee: FirstFit[3] as well as BestFit[4] have an asymptotic approximation ratio of 1.7, while their variants that consider items in decreasing order of size (called FirstFitDecreasing and BestFitDecreasing, respectively) achieve asymptotic ratios of 11/9 [77]. A big step forward was made by Fernandez de la Vega and Lueker [43], who gave an APTAS. Successively, Karmarkar and Karp [80] gave an AFPTAS for the problem. Their algorithm finds a solution with at most $\mathrm{OPT} + O\left(\log^2 \mathrm{OPT}\right)$ bins (where OPT denotes the optimal number of bins). Their approach was later improved by Rothvoß [97] to a polynomial time algorithm with a solution of size at most $\mathrm{OPT} + O(\log \mathrm{OPT} \cdot \log\log \mathrm{OPT})$ and by Hoberg and Rothvoß [67] to an algorithm with a solution of size at most $\mathrm{OPT} + O(\log \mathrm{OPT})$. It might still be possible that there exists an algorithm which returns a solution of size at most $\mathrm{OPT} + 1$.

In terms of absolute approximation ratio, Simchi-Levi [101] showed that FirstFitDecreasing and BestFitDecreasing have an absolute ratio of 3/2, which is best possible unless P=NP.

An interesting variant of bin packing are so-called "bounded space" algorithms: In such algorithms, at any point in time there is only a constant number of *open* bins, and items can only be packed into open bins (in contrast to *closed* bins to which no items will be added anymore). More specifically, an algorithm is k-bounded space if at any point there are at most k open bins. A natural k-bounded space variant of NextFit, called Next-k Fit, has an asymptotic approximation ratio of $1.7 + \frac{3}{10(k-1)}$ as showed by Mao [91], while the k-bounded space variant of BestFit is a 1.7-approximation for every $k \geq 2$ [32]. Chrobak et al. [26] showed that no 2-bounded space algorithm can have a better asymptotic ratio than 5/4 and gave a 3/2-approximative 2-bounded space algorithm. Note that their notion of approximation ratio compares the algorithmic solution with the best possible 2-*bounded space* solution.

Due to the broad applicability of the bin packing problem, there is a large variety of problem variants considered in the literature. Some of them are: cardinality constrained bin packing (at most k items can be packed into one bin) [3, 11], variable-sized bin packing (bins have different capacities and we want to minimize the total capacity of used bins) [39], bin packing with conflicts (some items might not be packed into the same bin) [37], bin packing with fragile objects (objects have a *fragility* and the sum of the sizes of the items in one bin must not exceed the minimum fragility among the items in this bin) [13], and many more.

[3] For each item, check all bins that were opened so far in the order they were opened and pack it into the first where it fits.

[4] For each item, find the first bin (in order of opening) with minimal unused capacity where it fits and pack it there.

Online version Bin packing was one of the very first problems considered in the online setting. The first bin packing algorithms were observed to have different behavior when sorting the input items in advance compared to when not doing this, leading to a very natural distinction between online and offline algorithms [76].

No 1-competitive algorithm for online bin packing can exist. This can be seen by a very simple example: Let two items of size 2/5 arrive. The algorithm has two possibilities: it either packs both items into the same bin or it packs them into separate bins. If it packs them into separate bins, we can end the input at this point and get a competitive ratio of 2 (the optimal solution would be using one bin to pack both). If the algorithm packs the items into one bin, we let two items of size 3/5 arrive. The algorithm needs to pack these into two new bins, using three bins in total, while the optimal solution only uses two bins (one small and one large item in each). That way, we see immediately that no online algorithm can find the optimal solution. This can of course be expanded to an input consisting of n items of either type, showing that in fact no online bin packing algorithm can have a competitive ratio of less than 4/3.

The question becomes now: Which is the best possible competitive ratio any online algorithm can achieve for this problem? We will now discuss some results about lower and upper bounds for this value.

The algorithms NextFit and FirstFit described previously are online algorithms by nature, giving upper bounds of 2 and 1.7, respectively. Johnson [76] posed the question whether any improvement upon this would be possible, which was answered by Yao [108] to the affirmative by providing the $5/3 \approx 1.666$-competitive algorithm Refined FirstFit. In 1985, Lee and Lee [86] proposed the so-called Harmonic algorithm, which achieved a competitive ratio of $(1 + \varepsilon)H_\infty$ for any $\varepsilon > 0$ where $H_\infty \approx 1.69103$. This algorithm is bounded space and Lee and Lee showed also that no algorithm which uses bounded space can perform better than H_∞. Later, Chrobak et al. [26] showed a lower bound of 4/3 for online 2-bounded space algorithms, comparing the online solution to the best possible offline 2-bounded space solution. Lee and Lee [86] also gave the (unbounded space) algorithm Refined Harmonic with competitive ratio less than 1.636.

After that, basically all published algorithms for the online bin packing problem were variations and improvements of Lee and Lee's Harmonic algorithm. Ramanan et al. [95] gave the algorithm Modified Harmonic with competitive ratio less than $\frac{538}{333} \approx 1.616\ldots$ and the algorithm Modified Harmonic 2 with competitive ratio $\approx 1.612\ldots$. They also showed a lower bound of $\frac{19}{12} \approx 1.58333$ for a larger class of algorithms including their algorithms, Harmonic and Refined Harmonic. In 2002, Seiden [98] unified all these algorithms in the framework Super Harmonic and within this framework specified the algorithm Harmonic++ which has competitive ratio at most 1.58889 and thus is very close to optimal within this framework of algorithms (the Ramanan et al. lower bound holds for it). After this work, no further improvements were made for almost 15 years prior to the work presented in this thesis.

Regarding general lower bounds, Yao [108] showed that no online algorithm can have a competitive ratio of less than 3/2, which was thereafter improved to 1.53635 by Brown [21] and Liang [88] independently, to 1.54014 by van Vliet [106] and finally to $\frac{248}{161} \approx 1.54037$ by Balogh et al. [8] where it stands today. The very slow progress in improving the lower bounds leads to our conjecture that this lower bound is much closer to the "real" best competitive ratio of an online algorithm than the current upper bounds.

Regarding absolute competitive ratio, the best known algorithm has a ratio of 5/3 which matches the lower bound [9].

Apart from the classical online setting, one might also consider so-called *semi-online* settings. In these, the online constraint is relaxed, meaning that the algorithm might have some information about future input or might be allowed to adjust its packing during processing. Several models have been considered. A very natural one is bin packing with repacking in which the algorithm is allowed to repack a certain (usually constant) number of items every time a new item arrives [49, 50, 52]. Another model, so-called batched bin packing, assumes that the input is split into parts by the adversary and each such part (*batch*) is given to the algorithm as one, allowing offline processing of this batch. This model was introduced by Gutin et al. [59].

Additionally, in recent years there has been an interest in bin packing with advice. This model investigates how many *bits* of additional information an online algorithm needs in order to achieve a certain competitive ratio; this also leads to the notion of advice complexity. To name one example, Boyar et al. [19] showed that the number of advice bits required to achieve an optimal solution is lower bounded by $(n - 2\mathrm{OPT}(\sigma))\log \mathrm{OPT}(\sigma)$ and upper bounded by $n\lceil \log \mathrm{OPT}(\sigma)\rceil$ (where $\mathrm{OPT}(\sigma)$ denotes the optimal number of bins needed to pack input σ). They also gave a 1.5-competitive algorithm with $\log n + o(\log n)$ bits of advice, a $(4/3+\varepsilon)$-competitive algorithm with $2n + o(n)$ bits of advice, and they showed that in order to achieve a competitive ratio better than 9/8, an online algorithms needs at least advice of linear size. For an overview about bin packing and other online problems with advice, see the recent survey by Boyar et al. [20].

3.2.2 Multi-dimensional bin packing

Offline version As soon as one turns to packing multi-dimensional items into bins, the problem becomes harder, as then it becomes even NP-hard to decide whether a given set of items can be packed into a single bin; this even holds in two dimensions when all items are squares [87]. This immediately implies that no absolute approximation ratio better than 2 is possible. Bansal et al. [14] showed that two-dimensional bin packing does not admit an APTAS unless P = NP.

Let us start by considering the two-dimensional case with arbitrary rectangles as input items.

When rotation of the items is not allowed, Kenyon and Rémila [82] gave a $(2+\varepsilon)$-approximation for any $\varepsilon > 0$. The important bound of 2 was broken by Caprara [22], who achieved an approximation ratio of $H_\infty \approx 1.69103$. For the case where rotations are allowed, there was a sequence of papers with approximation ratios 2.64 [93], $9/4 = 2.25$ [41], $2+\varepsilon$ [73] and $\ln(H_\infty)+1 \approx 1.525$ [15]. In 2013, Jansen and Prädel [70] gave an algorithm with approximation guarantee 1.5 for both cases (with and without rotation). Using their result, Bansal and Khan [12] showed that there is an approximation algorithm with asymptotic approximation ratio $\ln(1.5)+1+\varepsilon \approx 1.405+\varepsilon$ in both cases (with or without rotations) for any $\varepsilon > 0$. This is the best result known until today. On the other hand, the best known hardness of approximation results for two-dimensional bin packing are $1+1/3792$ (with rotation) and $1+1/2196$ (without rotation) [25], so the gap to the upper bound remains quite large. In the d-dimensional case with $d > 2$, the best known lower bound is H_∞^{d-1} [22].

For packing d-dimensional hypercubes, Kohayakawa et al. [83] gave a $(2-(2/3)^d+\varepsilon)$-approximation algorithm. Bansal et al. [14] improved this to an APTAS.

Regarding the absolute approximation ratio, as mentioned before, no ratio below 2 can be reached unless P = NP. Zhang [109] gave a 3-approximation for rectangles, while Harren and van Stee [63] gave a 3-approximation with better running time and a 2-approximation if rotations are allowed. Finally, a 2-approximation without rotation was shown by Harren et al. [64].

Online version An asymptotic 3.25-competitive algorithm for online bin packing in 2 dimensions was given by Coppersmith and Raghavan [30]. In three dimensions, they obtain a competitive ratio of 6.25. Csirik and van Vliet [33] gave an algorithm for arbitrary dimension d with competitive ratio H_∞^d, which amounts to roughly 2.859 for $d = 2$ and 4.835 for $d = 3$. This same bound was achieved by Epstein and van Stee [40] but with a bounded space algorithm; they also showed that no bounded space algorithm can improve this. They also give an algorithm for packing hypercubes. Interestingly, this algorithm can be shown to be optimal, however, the exact asymptotic competitive ratio is not known but only bounded by $\Omega(\log d)$ and $O(d/\log d)$. This open question was very recently resolved: Kohayakawa et al. [84] showed a lower bound of $\Omega(d/\log d)$ for bounded space hypercube packing algorithms. Seiden and van Stee [99] gave a 2.66013 competitive algorithm. Finally, the upper bound for two-dimensional bin packing was improved to 2.5545 by Han et al. [61]. The best known lower bound for two-dimensional online bin packing is 1.851 as given by van Vliet [107].

When rotations are allowed, Epstein [36] gave a $\frac{2935}{1152} + \delta \approx 2.54775$ competitive algorithm which is bounded space and also showed that this is almost optimal by proving a lower bound of $\frac{120754}{47628} \approx 2.53537$ for bounded space algorithms. She also gave an unbounded space algorithm with competitive ratio below 2.45.

Coppersmith and Raghavan [30] showed that no online algorithm can have competitive ratio better than 4/3 for square packing and gave a 2.6875 competitive algorithm (this is the same algorithm that has competitive ratio 3.25 in the general case). Seiden and van Stee [99] gave a 2.43828 competitive algorithm for online square packing, a lower bound of 2.28229 for bounded space online square packing, and an infinite lower bound for $d-1$-bounded space online hypercube packing (meaning that any such algorithm has arbitrary bad competitive ratio). Epstein and Stee [38] gave a 2.2697 competitive algorithm for square packing and a lower bound of 1.6406 for this problem. They also gave a 2.9421 competitive cube packing algorithm and a 1.668 lower bound for cubes. Han et al. [60] finally gave a 2.1187 upper bound for square packing and a 2.6161 upper bound for cube packing. These are the best bounds known. In the bounded space setting, Epstein and van Stee [42] gave lower and upper bounds for $d = 2, \ldots, 7$. In particular, for $d = 2$ the lower bound is 2.36343 and the upper bound 2.3692 and for $d = 3$ the lower bound is 2.95642 and the upper bound is 3.0672. Very recently, Balogh et al. improved the general lower bound for square packing to 1.75 [7].

3.3 Results in this thesis

In this thesis, we present several results on online bin packing in part I. Firstly, in chapter 4, we present two different one-dimensional online bin packing algorithms. These state the first improvement over the previous best algorithm in fifteen years and reduces the gap to the lower bound by over 15% and over 20%, respectively; both

results also beat the lower bound for the previous algorithm framework. We also show a lower bound for algorithms of our new framework.

Secondly, in chapter 5, we give lower bounds for multi-dimensional hypercube packing. We improve the general lower bound for square packing and rectangle packing in two dimensions. We also show a lower bound for Harmonic-type algorithms for hypercube packing in two or more dimensions, for the first time breaking the bound of 2, which uses a generalization of the method of Ramanan et al. [95]. Finally, we show that even when incorporating the ideas from chapter 4 that improved the one-dimensional case – essentially making use of knowledge about the items that already have arrived –, we obtain similar lower bounds as without these ideas. This indicates that further advances are necessary to obtain an improved two-dimensional algorithm.

4 Online One-Dimensional Bin Packing

In this chapter, we present two different one-dimensional online bin packing algorithms with asymptotic competitive ratio of 1.5813 and 1.5787, respecitvely. This beats the best algorithm known before which was Seiden's 1.58889-competitive Harmonic++ algorithm, and also beats the lower bound of 1.58333 by Ramanan et al. [95] for the whole class of algorithms in the Super Harmonic framework. In addition, we show how with a further improvement of Extreme Harmonic, we can reach a competitive ratio of 1.5787 and thus beat the bound of 1.58.

We make two crucial changes to the previous framework Super Harmonic. First, some of our algorithm's decisions depend on *exact sizes* of items, instead of only their types. In particular, for each item with size in $(1/3, 1/2]$, we use its exact size to determine if it can be packed together with an item of size greater than $1/2$. Second, we try to postpone some decisions made by the previous algorithm Super Harmonic to a later point in time in order to adapt to the input. We carefully mark items depending on how they end up packed, and use these marks to bound how many bins of a certain kind can exist in the optimal solution. This gives us better lower bounds on the optimal solution value. We show that for each input, a single weighting function can be constructed to upper bound the competitive ratio on it. We also give a lower bound of 1.5762 for this new framework.

For the second algorithm with competitive ratio 1.5787, we additionally handle the smallest type of items in the same way as all other items (packing them in two different ways), leading to a further improvement.

We also use our ideas to simplify the analysis of Super Harmonic, and show that the algorithm Harmonic++ is in fact 1.58880-competitive (Seiden proved 1.58889), and that 1.5884 can be achieved within the Super Harmonic framework.

Results in this chapter and organization First, we give an overview over the basic ideas and also the shortcomings of Super Harmonic in section 4.1. We then describe our new online bin packing algorithm framework, Extreme Harmonic; it is described and analyzed in section 4.2. In section 4.3, we describe a specific algorithm within this framework, called Son Of Harmonic, that achieves a competitive ratio of 1.5813. In section 4.4 we simplify and improve the analysis of Super Harmonic and Harmonic++. In section 4.5, we show that no algorithm within the Extreme Harmonic framework can give a competitive ratio below 1.5762. Finally, in section 4.6, we show that with only small modifications to Extreme Harmonic, we obtain a framework in which a competitive ratio of 1.5787 can be achieved.

4.1 Super Harmonic and its limitations

In 1985, Lee and Lee [86] described the Harmonic-algorithm, whose ideas became the basis for almost all online bin packing algorithms that followed. The central idea is the following: the interval $(0, 1]$ is partitioned into $m > 1$ intervals $I_1 = (1/2, 1], I_2 = (1/3, 1/2], \ldots, I_m = (0, 1/m]$, and the type of an item is then defined as the index of the interval which contains its size. Each type of items is packed into separate bins (i items per bin for type i), i.e., two items of different types will never be put into the same bin. This gives a very simple, bounded-space (we only have m open bins at all times, one for each type) algorithm. Lee and Lee [86] showed that for any $\varepsilon > 0$, there is a number m such that the Harmonic algorithm that uses m types has a competitive ratio of at most $(1 + \varepsilon)\Pi_\infty$, where $\Pi_\infty \approx 1.69103$.

To understand the biggest problem of Harmonic, which prevents a better competitive ratio, consider the following input sequence: First, n items of size $1/3 + \varepsilon$ arrive (for arbitrarily small, positive ε), followed by n items of size $1/2 + \varepsilon$. Harmonic will pack the two types of items separately, the first n items pairwise into $n/2$ bins, and the second n items separately into n additional bins. However, it is apparent that in bins with type 1 items, nearly half the space remains unused, and it would be much more prudent to pack one $1/3 + \varepsilon$ item with one $1/2 + \varepsilon$ item into each bin, using n bins in total. This means, one would use free space in bins with items of a certain type to pack items of *other* types there.

Unfortunately, this is not easy to achieve in the online setting. If the online algorithm simply packs the $1/3+\varepsilon$ items separately into bins in expectation of $1/2+\varepsilon$ items to follow, the adversary will simply end the input *without* sending $1/2 + \varepsilon$ items. In this case, the algorithm would use even twice as many bins as needed in the optimal solution. Therefore, one should try to find a middle ground: Pack *some* of the $1/3 + \varepsilon$ items alone into bins and the rest pairwise, in order to balance both possibilities (large items arrive or do not arrive).

It turns out that packing items of size in $(1/3, 1/2]$ and $(1/2, 2/3]$ carefully is crucial in order to achieve good competitive ratios. We therefore often use the notion of *medium items* for items with size in $(1/3, 1/2]$ and *large items* for items with size in $(1/2, 2/3]$.

A sequence of papers used the idea of combining items of different types to develop ever better algorithms [86, 95], and finally Seiden [98] presented a general framework called Super Harmonic which captures all of these algorithms.

Super Harmonic algorithms classify items based on an interval partition of $(0, 1]$ that is much more fine-grained than the one of Harmonic and give each item a color as it arrives, red or blue. For each type j, the fraction of red items is some constant denoted by red_j[1], and the item is colored upon arrival to maintain this constant as well as possible. Our goal is to combine in one bin red and blue items of different types, i.e., blue items of one type and red items of one other type, so that in each bin items of at most two different types are contained. In general, blue items should be packed as in Harmonic, i.e., $\lfloor 1/s_j \rfloor$ items of type j and size at most s_j will be packed in one bin. Red items should be packed only few per bin, intuitively they should not fill up more than $1/3$ of the bin (except for medium items, where red items are packed one per bin and of course fill up more than $1/3$ of the bin). Note that this also means

[1] This parameter was called α_j by Seiden; we have made many changes to the notation, which in many cases we feel was quite ad hoc.

that if red items of type i and blue items of type j are packed together into the same bin, then type i items will always be smaller than type j items.

If a new item arrives and is colored blue by the algorithm, we can either pack it with existing red items (we define later which types of red items are considered for this) or – if this is not possible – we pack it into a bin with only blue items of the same type. In such a bin, we hopefully can pack other red items later on. Similarly, we pack red items either with existing blue items or in new bins which are only partially filled. Hopefully, later blue items of another type arrive that can be placed into these bins with the red items. In the example from above, this would mean that some of the medium items (the red ones) are packed alone into a bin (the blue items, which are the majority, are still packed two per bin), so that *if* later large items arrive, they (or at least some of them) can be packed with the red medium items and *if no* large items arrive later we did not lose too much space as we only packed some medium items separately. One sees immediately that the ratio red_i that we chose for each type i immediately decides which of the two cases is worse and thus careful choice of these parameters is crucial in order to balance both cases and obtain a good overall competitive ratio.

Note that we speak of Super Harmonic as a *framework*, as it depends on many parameters (e.g. the interval partition that defines the types and the red_i-values); once we specify such a set of parameters, we obtain a concrete *algorithm*. The Super Harmonic algorithm Harmonic++, which uses 70 intervals for its classification and has about 40 manually set parameters (the red_i-values and others), achieves a competitive ratio of at most 1.58889 [98].

Ramanan et al. [95] gave a lower bound of $19/12 \approx 1.58333$ for this type of algorithm. One important type of inputs considered in their paper are inputs that contain a medium item and a large item[2]. Both of these items arrive N times for some large number N, and their sizes are carefully chosen for a specific Super Harmonic algorithm such that, although they fit pairwise into bins, the algorithm never combines them like this. See fig. 4.1 for such a pair of items for Harmonic++. No matter how fine the item classification of an algorithm, pairs of items such as these, that the algorithm does not pack together into one bin, can always be found. The result by Ramanan et al. shows that Harmonic++ is almost optimal within the Super Harmonic framework, and that some new ideas are required to overcome its shortcomings.

We now describe formally the Super Harmonic framework in section 4.1.1 and then we explain in more detail two main weakpoints of Super Harmonic algorithms in sections 4.1.3 and 4.1.4, in order to later show how our framework Extreme Harmonic, at least partially, overcomes these (section 4.2).

4.1.1 Formal definition of Super Harmonic

The fundamental idea of all Super Harmonic algorithms is to first classify items by size, and then pack an item according to its type (as opposed to letting the exact size influence packing decisions). For the classification of items, we use numbers $t_1 = 1 \geq t_2 \geq \cdots \geq t_N > 0$ to partition the interval $(0, 1]$ into subintervals $I_1, \ldots, I_N$. (N is a parameter of the algorithm.) We define $I_j = (t_{j+1}, t_j]$ for $j = 1, \ldots, N-1$ and $I_N = (0, t_N]$. We denote the type of an item $\mathfrak{p}$ by $t(\mathfrak{p})$, and its size by $s(\mathfrak{p})$. An item

[2] To complete the lower bound construction, they also consider inputs containing the sizes $1/3 + \varepsilon$, $1/2 + \varepsilon$, which can be combined into a single bin, and the input consisting only of items of size $1/3 + \varepsilon$.

$\mathfrak{p}$ has type j if $s(\mathfrak{p}) \in I_j$. A type j item has size at most t_j. We call the values t_j type thresholds (or simply thresholds if the context is clear).

Each item receives a color when it arrives, red or blue; an algorithm of the framework Extreme Harmonic defines parameters $\mathrm{red}_j \in [0,1]$ for each type j, which denotes the fraction of items of this type that are colored red. Blue items of type j are packed using NextFit: we use each bin until exactly $\mathrm{bluefit}_j := \lfloor 1/t_j \rfloor$ items are packed into it. For each bin, smaller red items may be packed into the space of size $1 - \mathrm{bluefit}_j t_j$ that remains unused. Red items are also packed using NextFit, using a fixed amount of the available space in a bin. This space is chosen in advance from a fixed set $\textsc{Redspace} = \{\mathrm{redspace}_i\}_{i=1}^{K}$ of spaces, where $\mathrm{redspace}_1 \leq \cdots \leq \mathrm{redspace}_K$. This determines the number of red items of type j that are packed together in one bin, which is denoted by redfit_j. In the space not used by red items, the algorithm may pack blue items. There may be several types that the algorithm can pack into a bin together with red items of type j. Each bin will contain items of at most two different types. If a bin contains items of two types, it is called mixed. If it contains items of only one type, but items of another type may be packed into this bin later, it is called unmixed. A bin that will always contain items of one type is called pure blue. A Super Harmonic algorithm tries to minimize the number of unmixed bins, and to place red and blue items in mixed bins whenever possible.

A Super Harmonic algorithm uses a function $\mathrm{leaves} : \{1,\ldots,N\} \to \{0,\ldots,K\}$ to map each item type to an index of a space in Redspace, indicating how much space for red items it leaves unused in bins with blue items of this type. Here $\mathrm{leaves}(j) = 0$ means that no space is left for red items. The algorithm also uses a function $\mathrm{needs} : \{1,\ldots,N\} \to \{0,\ldots,K\}$ to map how much space (given by an index of Redspace) red items of each type require. We have $\mathrm{needs}(i) = 0$ if and only if $\mathrm{red}_i = 0$ (i.e., there are no red items of this type).

The *class* of an item of type t is $\mathrm{leaves}(t)$, if it is blue, and $\mathrm{needs}(t)$ if it is red. The class of an item $\mathfrak{p}$ indicates how much space is reserved for red items in the bin containing $\mathfrak{p}$ (both if $\mathfrak{p}$ is red and if $\mathfrak{p}$ is blue), namely $\mathrm{redspace}_i$ space if the class is i.

For each type i such that $\mathrm{leaves}(i) = 0$, the items of this type are packed in pure blue bins, that contain only blue items (only one type per bin). An unmixed bin is called unmixed blue or unmixed red depending on the color of the items in it.

A mixed bin with blue items of type i and red items of type j satisfies the following properties: $\mathrm{leaves}(i) > 0, \mathrm{red}_j > 0, \mathrm{needs}(j) > 0$, and $\mathrm{redspace}_{\mathrm{needs}(j)} \leq$

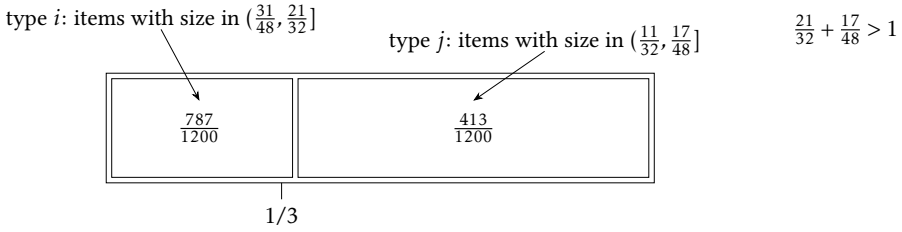


Figure 4.1: A critical bin. The item sizes are chosen such that a given Super Harmonic algorithm (in this case, Harmonic++) does not pack these items together. For any Super Harmonic algorithm, such sizes can be found. The central idea of our new algorithm is that we limit the number of times that these critical bins can occur in the optimal solution. This is how we beat the ratio of 1.58333.

$\text{redspace}_{\text{leaves}(i)}$. Observe that the last inequality holds if and only if $\text{needs}(j) \leq \text{leaves}(i)$. The blue items will use space at most $1 - \text{redspace}_{\text{leaves}(i)}$ and the red items will use space at most $\text{redspace}_{\text{needs}(j)} \leq \text{redspace}_{\text{leaves}(i)}$.

Definition 8. *An unmixed blue bin with blue items of type j is* compatible *with a red item of type i if* $\text{needs}(i) \leq \text{leaves}(j)$. *An unmixed red bin with red items of type j is compatible with a blue item of type i if* $\text{needs}(j) \leq \text{leaves}(i)$.

In both cases, the condition means that the blue items and the red items together would use at most 1 space in the bin (the blue items leave enough space for the red items and vice versa).

Definition 9. *A bin is* red-open *for a non-sand item of type t if it contains at least one and at most* $\text{redfit}_t - 1$ *red t items. A bin is* blue-open *for a non-sand item of type t if it contains at least one and at most* $\text{bluefit}_t - 1$ *blue t items. A bin is* open *if it is red-open or blue-open.*

Red-open bins with red items of type j contain at least one and at most $\text{redfit}_j - 1$ red items. Blue-open bins can be pure blue. Red-open and blue-open bins can be mixed or unmixed. Mixed bins can be red-open and blue-open at the same time. A bin with bluefit_i items of type i but no red items is not considered open, even though red items might still be packed into it later.

A formal definition of Super Harmonic in pseudo-code is given in algorithm 1. This is a much more compact representation than the one Seiden used, following the representation later used for our framework.

Note about sand items Items of size at most t_N are called *sand*. As in Harmonic, such items are packed completely independent from all other item types (in Super Harmonic as well as in our framework Extreme Harmonic). We do not use parameters such as bluefit for this type, as such items are packed using Any Fit into separate bins. They are also not explicitly mentioned in, e.g., algorithm 1 to keep the presentation simpler.

However, the framework Extreme Harmonic can easily be extended to also color sand items in a similar way as all other items; this extension and the resulting competitive ratio are discussed in section 4.6.

Algorithm 1 How the Super Harmonic framework packs a single item $\mathfrak{p}$ of type $i \leq N - 1$. At the beginning, we set $n_r^i \leftarrow 0$ and $n^i \leftarrow 0$ for $1 \leq i \leq N - 1$.

```
1: n^i ← n^i + 1
2: if n_r^i < ⌊red_i n^i⌋ then                    // pack a red item
3:     PackSH(p, red)
4:     n_r^i ← n_r^i + 1
5: else                                           // pack a blue item
6:     PackSH(p, blue)
7: end if
```

Algorithm 2 The algorithm PackSH($\mathfrak{p}, c$) for packing an item $\mathfrak{p}$ of type i with color $c \in$ {blue, red}.

1: Try the following types of bins to place $\mathfrak{p}$ with color c in this order:
2: • a pure blue, mixed, or unmixed c-open bin with items of type i and color c
3: • an unmixed bin that is *compatible* with $\mathfrak{p}$ (the bin becomes mixed)
4: • a new unmixed bin (or pure blue bin, if leaves$(i) = 0$ and $c =$ blue)

4.1.2 Analyzing Super Harmonic using weighting functions

For anaylizing bin packing algorithms, one of the best-known and most used approaches are weighting functions ([104, 86, 95]). The high-level idea is to define one or more weighting functions that assign each item a weight such that the average weight per bin in the algorithm's solution is one. Then, we can analyze the maximum weight that can be packed into a single bin: This gives an upper bound on the competitive ratio.

Seiden generalizes the previously used simpler concept of weighting functions to weighting systems. A weighting system is basically characterized by a dimension m and two functions $w : (0, 1] \mapsto \mathbb{R}^m$ and $\xi : \mathbb{R}^m \mapsto \mathbb{R}$. w is the weighting function which assigns each item of size s a set of m weights $w(s)_1, \ldots, w(s)_m$. ξ is called consolidation function. In order for w and ξ to describe a feasible weighting system for algorithm $\mathcal{A}$, the inequality $\mathcal{A}(\sigma) \leq \xi(\sum_{i=1}^{N} w(s(p_i))) + O(1)$ has to hold, where σ is an arbitrary input sequence and p_i is the i-th item in this sequence. One can reformulate the weighting system Seiden used in such a way that one obtains two weighting functions that assign each item one weight for each of $K+1$ cases (where K is the number of redspace_i-values). The consolidation function then basically takes the minimum of the two functions and maximizes this over the $K+1$ cases.

Now finding this maximum would give the desired upper bound on the competitive ratio. This is basically a one-dimensional knapsack problem, however, using two weighting functions per item. Seiden developed a heuristic approach to solve this knapsack problem in a feasible time (roughly 36 hours) using a computer program.

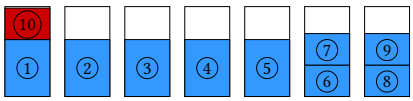
4.1.3 The Ramanan et al. lower bound

Let us take a closer look into the lower bound construction of Ramanan et al. [95], to gain more insight into what prevents Super Harmonic-algorithms from performing better. The class of algorithms considered by Ramanan et al. provide, using some parameter $h \geq 1$, a partition of the interval $[0, 1]$ into disjoint subintervals, including $I_{1,j} = (1 - y_{j+1}, 1 - y_j]$ and $I_{2,j} = (y_{h-j}, y_{h-j+1}]$ for $j = 1, \ldots, h$ with parameters $1/3 = y_0 < y_1 < \ldots < y_h < y_{h+1} = 1/2$ and requires that items of two subintervals $I_{1,j}$ and $I_{2,k}$ are only packed into the same bin if $j + k > h$. One of the inputs considered in the lower bound proof consists of items of size $1 - y_{h-j+1} + \varepsilon$ (i.e., at the lower end of type $I_{1,h-j}$) and items of size $y_{h-j} + \varepsilon$ (i.e., at the lower end of type $I_{2,j}$)[3]. The parameter ε is chosen in such a way that these two items fit together into one bin, however, as the algorithm only judges items based on their *type*, not on their *exact size*, it handles these items as if they had *maximum* size within their type. That is, the items are handled as if they have size $1 - y_{h-j}$ and y_{h-j+1} (one can think of them being rounded

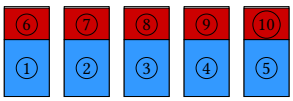
[3]The approach by Ramanan et al. is generalized in section 5.3, see there for more details on the construction.

up within their type), which makes them not fitting together into one bin. Hence, the algorithm does not pack the items into one bin although they would actually fit. We call such bins *critical*, see also fig. 4.1.

It is the natural approach to overcome this problem by just combining items whenever they fit together, ignoring their type. However, if we simply add this rule to the Super Harmonic framework, we get an additional problem: It might be impossible now to maintain the fixed fraction of red items red_i, which is however crucial for the analysis. The problem can occur if, e.g., the algorithm is first given n large items of size $1/2 + \varepsilon$, followed by n medium items of size $1/3 + \varepsilon$. According to our new rule, the algorithm would combine one large and one medium item in each bin, and obviously, this is exactly the right way to go. However, now all of the medium items are red, while our ratios red_i for the medium types are of course substantially below one. Known methods for analyzing such an algorithm, however, heavily depend on the fact that these ratios are maintained exactly. On the other hand, if we stick with the fixed fraction red_j for the medium items in such a case we end up with the same worst case instances as for Super Harmonic (see fig. 4.2).



(a) Packing produced by Super Harmonic.



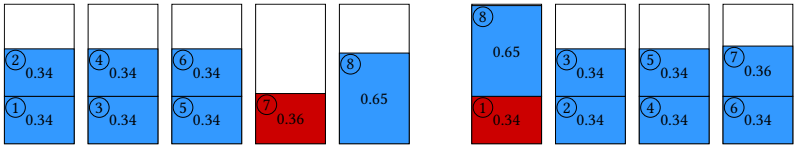
(b) Optimal packing and packing produced by Extreme Harmonic.

Figure 4.2: An example illustrating why it might be necessary to color more than a red_i-fraction of the items of a medium type i red. First, five large items arrive (numbered 1-5), then five medium items of type i (numbered 6-10). We assume that $\text{red}_i = 1/5$ here.

Finally, even if we add the rule to combine items whenever they fit together, there is still another problem that might prevent the algorithm from combining items in cases where the optimal solution is able to do so; this is discussed in the following.

4.1.4 The size of red items

Imagine the following situation: We have a type that contains items of size in $(1/3, 0.37]$, and we want to color $1/7$ of these items red (i.e., red_i for this type is $1/7$). Super Harmonic would do this by simply coloring every seventh item red and all other items blue. Now imagine that we would combine large items with medium items whenever they fit together (which Super Harmonic does not do, as discussed in the previous subsection). Consider an input that consists of six items of size 0.34, followed by one item of size 0.36 and one item of size 0.65; for the sake of asymptotic analysis, repeat this input arbitrarily often. Super Harmonic will color the 0.34 items blue and pack them pairwise into three bins, the 0.36 item will be colored red and packed alone into a bin, and when the large item arrives, the algorithm will not be able to combine it with the red item, as this is too large. However, if the algorithm would have colored one of the 0.34 items red and the 0.36 item blue, it would have been possible to combine the red medium and the large item (and the offline optimum of course does so). A weakness of all Super Harmonic algorithms is that they do not distinguish between any two items that have the same type. See also fig. 4.3 for an illustration.



(a) Packing produced by Super Harmonic. (b) Optimal packing.

Figure 4.3: Packing of a sequence of medium items (all of the same type i) followed by one large item. The items arrive in the order indicated by the numbers in circles. Super Harmonic needs more bins than the optimal solution, as the red medium item is too large to be combined with the large item. (We assume $\mathrm{red}_i = 1/7$ here.)

It becomes clear that we should try to assure that the *smallest* items of each type become red. However, this is not easy to attain in the online setting: When an item arrives, we do not know whether, within its type, we will encounter larger or smaller items in the future, thus whether it would be better to color it red or blue. However, as we immediately have to decide where to pack the item, and this also means whether to combine it with possibly existing other blue items of the same type or red items of some other type, we need to color the item upon arrival. We will see in the next section how this problem can be overcome – at least to some extent – by carefully postponing the coloring decision.

4.2 A new framework: Extreme Harmonic

4.2.1 Overview

We first describe on a high level how our framework Extreme Harmonic overcomes the problems of Super Harmonic mentioned before.

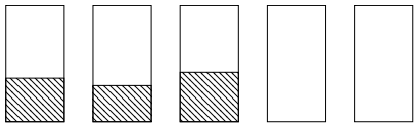
Combining items depending on exact size

First of all, our algorithm will combine large and medium items *whenever* they fit together, ignoring their type. Essentially, we use Any Fit to combine such items into bins (under certain conditions specified below). This is a generalization of the well-known algorithms FirstFit and BestFit [104, 56], which have been used in similar contexts before [9, 4]. For all other items, we essentially leave the structure of Super Harmonic intact, although a number of technical changes are made, as we describe next. We maintain the property that each bin contains items from at most two types, and if there are two types in a bin, then the items of one type are colored blue and the others are colored red.

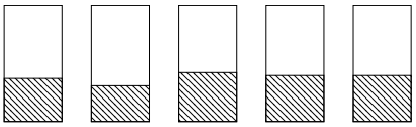
Provisional coloring

As discussed above, in order to benefit from using Any Fit, it is important to ensure that for each medium type, as much as possible, the *smallest items* are colored red. In order to do this, we initially pack *each* medium item alone into a bin *without assigning a color*. After several items of the same type have arrived, we will color the smallest one red and the others blue and start packing additional medium items of the same type together with these blue items (see fig. 4.4.) In this way, we can ensure that at

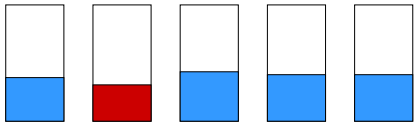
least half of the blue items (namely, the ones that have already arrived at the time when we select the smallest one to be red) are at least as large as the smallest red item.



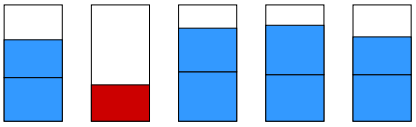
(a) Pack items one per bin without coloring them.



(b) The fifth item arrives: time to fix the colors.



(c) The smallest item becomes red.



(d) Additional blue items of the same type are added.

Figure 4.4: Illustration of the coloring in Extreme Harmonic. Hatched items are uncolored. In this example, $\text{red}_i = 1/9$, where i is the type of all items depicted in this example. Note that the ratio of 1/9 does not hold (for the bins shown) at the time that the colors are fixed: 1/5 of the items are red at this point. The ratio 1/9 is achieved when all bins with blue items contain two blue items. The blue items which arrive in step (d) are called *late* items.

However, postponing coloring decisions like this is not always possible or even desirable. In fact there are exactly two cases where this will not be done upon arrival of a new medium item p.

1. If a bin with suitable small red items (say, of some type t) is available, and it is time to color p blue, we will pack p into that bin and color it blue, regardless of the precise size of p. In this case, in our analysis we will carefully consider how many small items of type t the input contains; knowing that there must be some. This implies that in the optimal solution, not *all* the bins can be critical. Moreover, our algorithm packs these small items very well, using almost the entire space in the bin.

2. If a bin with a large item is available, and p fits into such a bin, we will pack p in one such bin as a red item regardless of which color it was supposed to get. This is the best case overall, since finding combinations like this was exactly our goal! In fact we must pack p like this, else we end up with the same worst case instances as for Super Harmonic (fig. 4.2). However, there is a technical problem with this, which we discuss below.

Overall, we have three different cases: medium items are packed alone initially (in which case we have a guarantee about the sizes of some of the blue items), medium items are combined with smaller red items (in which case these small items exist and must be packed in the optimal solution), or medium items are combined with larger blue items (which is exactly our goal).

The main technical challenge is to quantify these different advantages into one overall analysis. In order to do this, we introduce – in addition to and separate from the coloring – a marking of the medium items. The marking indicates whether the

blue or red items of a given mark are in mixed or unmixed bins. This will bound the number of critical bins (fig. 4.1) that can exist in the optimal solution, leading to better lower bounds for the optimal solution value than Seiden [98] used.

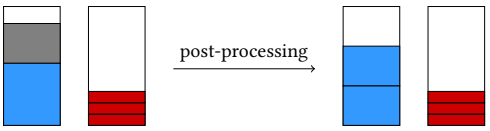
Post-processing and change of input

Maintaining the fraction red_j of red items for all marks separately is necessary for the analysis. As we have seen however, if many large items arrive first, we must pack medium items with them whenever possible, even if this violates the ratio red_j. If there are more than red_j medium items of some type j when the input ends, we call those items *bonus items*. Each bonus item is packed in a bin with a large item. After the input ends, we will (virtually) make some of those large items *smaller* so that they get type j as well (see fig. 4.5a). We then change the colors of the bonus items to ensure the proper fraction red_j of red medium items. Hence we *modify the input*, but we only do this for the analysis and only once all the items have been packed. Clearly the number of bins in the optimal solution can only decrease as a result of making some items smaller. In order to prove that the modification is legal, we additionally have to argue that our analysis using weighting functions still counts correctly the number of bins *after* the modifications were applied (i.e., in particular, we do not consider the algorithm's solution on the modified input directly; this indeed would first require fixing an order for the items to arrive in).

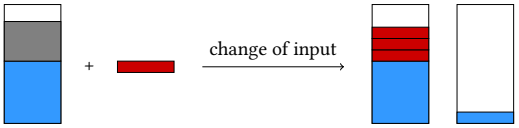
However, this is not the case if we handle bonus items simply like that; consider for illustration fig. 4.5a. The problem is that there could be small red items (say, of type t) in separate bins that could have been packed in bins with two medium type j items, had such bins been available at the time when the small red items arrived. Creating such bins after the input ends generates a packing that would not be analyzed correctly by our weighting functions. This would make our analysis invalid. To avoid this, we *do not allow* small items to be packed into new bins as red items as long as bins with large and medium items exist that may later be modified. Instead, in such a case, we count a single medium item in such a bin as a number of red small items of type t, and pack the incoming item of type t as a blue item (fig. 4.5b). This ensures (as we will show) that if suitable bins with blue items are available, red items of type t are always packed in them, rather than in new bins.

Analysis

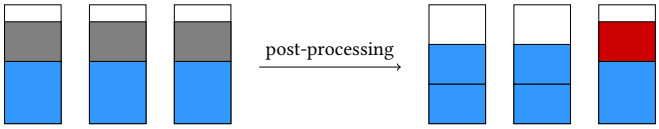
When analyzing Seiden's algorithm with the weighting function technique, it turns out that bins with a medium and a large item, i.e. those bins that are used in the Ramanan et al. lower bound and called critical in this thesis, have the highest weight and therefore determine the competitive ratio. In order to gain some advantage from our techniques, we have to do a more complicated analysis: We do not assume that all bins in the optimal solution are packed with maximum weight, as this is an overly pessimistic assumption. Instead, we analyze in a more detailed fashion which bins need to occur in the optimal solution and can then give an upper bound on the competitive ratio that results from the *average* weight in the optimal solution's bins. Seiden set up a mathematical program that determined the competitive ratio and then used a heuristic approach to solve this huge LP efficiently. As we add more constraints to the LP, this heuristic approach does not carry over to our setting immediately, so instead we split the mathematical program into two linear programs and then consider the dual of these LPs and show how to solve it using the ellipsoid method.



(a) This situation **cannot occur** in our algorithm: We shrink a large item packed with a bonus item but there are uncombined red small items compatible with the bonus item.



(b) In order to prevent the situation in fig. 4.5a, we (virtually) resize and split the bonus item into small items when other small red items arrive. The new item becomes blue instead. Later, more small blue items can be packed with it.



(c) Only when bonus items remain after the algorithm terminates do we transform some large items to medium items, re-establishing the correct ratio of red items for the medium items (in this example, this ratio is 1/5).

Figure 4.5: Post-processing and change of the input for the analysis. Gray items denote bonus items.

We have to implement the ellipsoid method in order to be able to give a value for the competitive ratio. We do this by writing the dual in terms of just one variable, by eliminating two variables and assuming a third one to be given. This means that we can now do a straightforward binary search for the final remaining variable. We implemented a computer program which solves the knapsack problems and also does the other necessary work, including the automated setting of many parameters like item sizes and values red_i. As a result, our algorithm Son Of Harmonic requires far less manual settings than Harmonic++.

Our program uses an exact representation of fractions, with numerators and denominators of potentially unbounded size, in order to avoid rounding errors. For our final calculations we have set the bound such that every dual LP is feasible; this means that our results do not rely on the correctness of any infeasibility claims (which are generally harder to prove). We also provide the final set of knapsack problems directly to allow independent verification.

This approach can also be applied to the original Super Harmonic framework. Surprisingly, we find that the algorithm Harmonic++ is in fact 1.58880-competitive. We suspect that Seiden did not prove this ratio because of the prohibitive running times of his heuristic approach; he mentions that it took 36 hours to prove the upper bound of 1.58889. Our program completes in few seconds. Another benefit of using our approach is that this result becomes more easily verifiable as well. Furthermore, we were able to improve and simplify the parameters of Harmonic++ to obtain a competitive ratio of 1.5884. These results are discussed in section 4.4.

4.2.2 The algorithm

The Extreme Harmonic framework

Let us first formally define the different item size ranges.

Definition 10. *An item is called*

- huge *if its size is in* $(2/3, 1]$,
- large *if its size is in* $(1/2, 2/3]$,
- medium *if its size is in* $(1/3, 1/2]$, *and*
- small *if its size is at most* $1/3$

Note that from now on, there will be no types that contain $1/2$ or $1/3$ as an inner point in their interval, and thus, we can extend the notion of huge, large, medium and small items to types in the natural way.

Next, we extend the definition of compatible bins (definition 8) as follows. As noted before, some items will not receive a color when they arrive, but only later. The goal of having uncolored items is to try and make sure that relatively small items of each medium type become red in the end (to make it easier to combine them with large items).

Definition 11. *An unmixed bin is* red-compatible *with a newly arriving item* $\mathfrak{p}$ *of type* $\leq N-1$ *if*

1. *the bin contains blue or uncolored items*[4] *of type* i *with* $\text{leaves}(i) \geq \text{needs}(t(\mathfrak{p}))$, or
2. *the bin contains a (blue) large item of size* x, $\mathfrak{p}$ *is medium and* $s(\mathfrak{p}) \leq 1 - x$.

An unmixed bin is blue-compatible *with a newly arriving item* $\mathfrak{p}$ *if*

1. *the bin contains red items*[4] *of type* j *and* $\text{leaves}(t(\mathfrak{p})) \geq \text{needs}(j)$, or
2. *the bin contains one red or uncolored medium item of size* x, $\mathfrak{p}$ *is large and* $s(\mathfrak{p}) \leq 1 - x$.

For checking whether a large item and a medium item can be combined in a bin, we ignore the values $\text{leaves}(i)$ and $\text{needs}(j)$ and use only the relevant parts 2 of definition 11.

Recall that a bin is red-open (blue-open) if it contains some red items (blue items) but can still receive additional red items (blue items). Like Super Harmonic algorithms, an Extreme Harmonic algorithm first tries to pack a red (blue) item into a red-open (blue-open) bin with items of the same type and color; then it tries to find an unmixed compatible bin; if all else fails, it opens a new bin. Note that the definition of compatible has been extended compared to Super Harmonic, but we still pack blue items with red items of another type and vice versa; there will be no bins with blue (or red) items of two different types. The new framework is formally described in algorithms 3 and 4. Items of type N are packed using NextFit as before. We discuss the changes from Super Harmonic one by one. All the changes stem from our much more careful packing of medium items. The algorithm Mark and Color called in

[4] We will see later that if an item has no color, it is the only item in its bin (property 7).

line 27 of Extreme Harmonic will be presented in section 4.2.4. This algorithm will take care of assigning marks and colors to the items. In particular, this will take care of fixing the color of medium items as described in fig. 4.4.

As can be seen in Pack (lines 2, 4 and 5), medium items that are packed into new bins are initially packed **one** per bin and not given a color. We wait until enough of these items have arrived, and then color the smallest one red using Mark and Color (fig. 4.4). Note that n_r^i is increased in line 16 of algorithm 3 even though the item might not receive a color at this time. This means that the value n_r^i does not alway accurately reflect how many red items there are currently. We will show that this is not an issue for the analysis (it will be accurate up to a constant).

Algorithm 3 How the Extreme Harmonic framework packs a single item $\mathfrak{p}$ of type $i \le N-1$. At the beginning, we set $n_r^i \leftarrow 0$, $n_{\text{bonus}}^i \leftarrow 0$ and $n^i \leftarrow 0$ for $1 \le i \le N-1$.

1: $n^i \leftarrow n^i + 1$
2: **if** $n_r^i < \lfloor \text{red}_i n^i \rfloor$ **then** // *pack a red item*
3: **if** $n_{\text{bonus}}^i > 0$ or $\text{needs}(i) \le 1/3 \wedge \exists j : n_{\text{bonus}}^j > 0$ **then**
// *special case: replace bonus item and make the new item blue; see fig. 4.5c*
4: **if** $n_{\text{bonus}}^i > 0$ **then**
5: Let $\mathfrak{b}$ be a bonus item of type i // *in this case,* $\text{redfit}_i = 1$
6: **else**
7: Let $\mathfrak{b}$ be a bonus item of some type j with $n_{\text{bonus}}^j > 0$ // *here i is a small type*
8: **end if**
9: $n_{\text{bonus}}^{t(\mathfrak{b})} \leftarrow n_{\text{bonus}}^{t(\mathfrak{b})} - 1$
10: Label $\mathfrak{b}$ as type i // *count $\mathfrak{b}$ as type i item(s) and color it/them red*
11: $n^i \leftarrow n^i + \text{redfit}_i$ // *$\mathfrak{b}$ might have been of type i already, then* $\text{redfit}_i = 1$
12: $n_r^i \leftarrow n_r^i + \text{redfit}_i$
13: Pack($\mathfrak{p}$, blue) // *since we now have $n_r^i \ge \lfloor \text{red}_i n^i \rfloor$ again*
14: **else**
15: Pack($\mathfrak{p}$, red)
16: $n_r^i \leftarrow n_r^i + 1$ // *The item is red or uncolored*
17: **end if**
18: **else** // *pack a blue item*
19: **if** $\mathfrak{p}$ is medium, $\text{red}_i > 0$, and there exists a bin B that is red-compatible with $\mathfrak{p}$ **then**
20: Place $\mathfrak{p}$ in B and label it as bonus item. // *special case: bonus item*
21: $n^i \leftarrow n^i - 1$ // *we do not count this item for type i*
22: $n_{\text{bonus}}^i \leftarrow n_{\text{bonus}}^i + 1$ // *Note that B contains a large item*
23: **else**
24: Pack($\mathfrak{p}$, blue) // *The item is blue or uncolored*
25: **end if**
26: **end if**
27: Update the marks and colors using Mark and Color (section 4.2.4).

When an item arrives, in many cases, we cannot postpone assigning it a color, since a c-open or c-compatible bin is already available (see lines 2 to 3 of Pack(p,c)). Additionally, if we are about to color a medium item blue because currently $n_r^i \ge \lfloor \text{red}_i n^i \rfloor$, we check whether a suitable large item has arrived earlier. We deal with this case in lines 19 to 22 of Extreme Harmonic. In this special case, we *ignore* the

Algorithm 4 The algorithm Pack($\mathfrak{p}, c$) for packing an item $\mathfrak{p}$ of type i with color $c \in \{\text{blue}, \text{red}\}$.

1: Try the following types of bins to place $\mathfrak{p}$ with (planned) color c in this order:
2: • a pure blue, mixed, or unmixed c-open bin with items of type i and color c
3: • a c-compatible unmixed bin (the bin becomes mixed, with fixed colors of its items)
4: • a new unmixed bin (or pure blue bin, if $\text{leaves}(i) = 0$ and $c = \text{blue}$)
5: If $\mathfrak{p}$ was packed into a new bin, $\mathfrak{p}$ is medium and $\text{red}_i > 0$, give $\mathfrak{p}$ **no** color, else give it the color c.

value red_i. We pack the medium item with the large item as if it were red (no further item will be packed into this bin), but we *do not count* it towards the total number of existing medium items of its type; instead we label it a *bonus item*. Bonus items do not have a mark or color, but this can change later during processing in the following two cases.

1. Additional items of type i arrive which are packed as blue items. If enough of them arrive (so that it is time to color an item red again), we first check in line 3 of Extreme Harmonic if there is a bonus item of type i that we could color red instead. If there is, we will do so, and pack the new item as a blue item.

2. An item of some type j and size at most 1/3 arrives, that should be colored red. In this case, for our accounting, we view the bonus item as redfit_j red items of type j, and adjust the counts accordingly in lines 10 to 12 of Extreme Harmonic.[5] The new item of type j is packed as a blue item in line 13 of Extreme Harmonic.[6]

It can be seen that blue items of size at most 1/3 are packed as in Super Harmonic. For red items of size at most 1/3, we deal with existing bonus items in lines 10 to 12 of Extreme Harmonic, and in line 3 of Pack($\mathfrak{p}, c$), an existing medium item may be colored red or blue (the opposite of the parameter c). Otherwise, the packing proceeds as in Super Harmonic for these items as well.

Parameter requirements

In order to make our analysis work for all Extreme Harmonic algorithms, we require their parameters to fulfill certain conditions, which are listed now.

R(1) If j is a small type with $\text{red}_j > 0$, $\text{redspace}_{\text{needs}(j)} \leq 1/3$.

R(2) If i is a large or huge type, then $\text{red}_i = 0$, so $n_r^i = 0$ at all times.

R(3) For $x > 1/3$, we have $x \in \text{Redspace}$ if and only if $\exists i : x = t_i$ and i is medium.

R(4) We have $\text{red}_i < 1/3$ for all types i.

R(5) We have $t_1 = 1, t_2 = 2/3, t_3 = 1/2$, and $\text{red}_1 = \text{red}_2 = \text{red}_N = 0$.

[5] Note that the meanings of i and j are switched in the description of the algorithm for reasons of presentation.

[6] Strictly speaking, we only need this whole procedure if type j is compatible with the bonus item, to avoid the case in fig. 4.5b. Instead, we do it for all small items for simplicity.

R(6) All type 1 items (i.e., huge items) and type N items are packed in pure blue bins. Equivalently, $\text{leaves}(1) = \text{leaves}(N) = 0$.

R(7) We have $1/3 \in$ Redspace.

R(8) Let t_i, t_{i+1} be two consecutive medium type thresholds of the algorithm. Then $t_i - t_{i+1} < t_N < 1/100$.

Note that requirement R(3) implies that $\text{redfit}_i = 1$ for medium types i.

4.2.3 Properties of Extreme Harmonic algorithms

We now prove (or state, for easily observable ones) some properties of Extreme Harmonic algorithms. Let $\varepsilon = t_N$.

Property 1 (Lemma 2.1 in Seiden [98])**.** *Each bin containing items of type N, apart from possibly the last one, contains items of total size at least $1 - \varepsilon$.*

Property 2. *For any type i, if* $\text{needs}(i) > 0$*, then* $\text{leaves}(i) < \text{needs}(i)$.

Proof. If $\text{needs}(i) > 0$, then $\text{red}_i > 0$. If $\text{leaves}(i) \geq \text{needs}(i) > 0$, an additional item of type i could be placed in the space $\text{redspace}_{\text{needs}(i)} \leq \text{redspace}_{\text{leaves}(i)}$, which means we could fit $\text{bluefit}_i + 1$ blue items of type i into one bin, contradicting the definition of bluefit_i. □

Property 3. *For a medium type i, if* $\text{leaves}(i) > 0$ *then* $\text{redspace}_{\text{leaves}(i)} < 1/3$*, and if* $\text{red}_i > 0$*, then* $1/3 < \text{redspace}_{\text{needs}(i)}$.

Requirements R(5) and R(7) together imply that $\text{redspace}_{\text{leaves}(2)} = 1/3$. This means together with property 3 that an unmixed bin with a large item is never red-compatible with a medium item via condition 1 of definition 11 (so only condition 2 is relevant for this combination). This furthermore implies that for a medium item of type i, the precise value $\text{needs}(i)$ is irrelevant for the algorithm (only the fact that $\text{redspace}_{\text{needs}(i)} > 1/3$ is relevant). It will nevertheless be useful for the analysis to have $t_i \in$ Redspace as required by requirement R(3).

Property 4. *For each type i and color c, there is at most one c-open bin that contains items of type i and no other type. For each pair of types and color c, there is at most one c-open bin with items of those types.*

Proof. Consider an item of type i and color c. By the order in which Pack tries to place items into bins, we only open a new unmixed or pure blue bin of type i if no c-open bin is available, so the first claim holds.

Now consider a pair of types. Say the blue items are of type i and the red items are of type j. The only cases in which a mixed bin with such items is created are the following:

- A red item of type j is placed into an unmixed bin B with blue items of type i. In this case, there was no existing red-open mixed or unmixed bin with red items of type j.
- A blue item of type i is placed into an unmixed bin B with red items of type j. In this case, there was no existing blue-open mixed, unmixed or pure blue bin with blue items of type i.

- A bin receives a bonus item in line 20 of Extreme Harmonic and is now considered mixed.
- A bonus item gets counted as items of type j in lines 10 to 12.

At the beginning, there are zero open bins with items of type i and type j. Such bins are only created via one of the cases listed above. In the last two cases, no open bins are created (note that only one medium and one large item can be packed together in a bin). In the second case, B is the only red-open bin with these types (if $\text{redfit}_j > 1$, that is), and no red items of type j are packed into unmixed bins with blue items until B contains redfit_j type j items by line 2 of algorithm 4. In the first case, similarly, no new blue item of type i will be packed into unmixed bins with red items as long as B remains blue-open. □

Property 5. *At all times, for each type i, $n_r^i \geq \lfloor \text{red}_i n^i \rfloor - 1$. For each medium type i, $n_r^i \leq \lfloor \text{red}_i n^i \rfloor$. For each small type i, $n_r^i \leq \lfloor \text{red}_i n^i \rfloor + \text{redfit}_i$.*

Proof. The first bound follows from the condition in line 2 of algorithm 3 and because n^i increases by at most 1 in between two consecutive times that this condition is tested, unless lines 10 to 12 of Extreme Harmonic are executed; but in that case, the fraction of red items of type i only increases, because n^i and n_r^i increase by the same amount.

The upper bounds follow because for each medium type i, n_r^i increases by at most 1 when $n_r^i < \lfloor \text{red}_i n^i \rfloor$ and a new item of this type arrives: either in line 12 ($\text{redfit}_i = 1$ for medium items) or in line 16. Furthermore, if $n_r^i = \lfloor \text{red}_i n^i \rfloor$, n_r^i is not increased anymore. For small items, n_r^i increases by at most redfit_i in one iteration (line 12), and this only happens if the ratio is too low (line 2). □

Recall that n_r^i is not always the true number of medium red items of type i, as some of these may not have a color yet. For a small type i, the value n_r^i may also not be accurate, because it may include some bonus items. We will fix this in postprocessing, where we replace the bonus items by items of type i to facilitate the analysis.

Property 6. *At all times, for each type i that is not medium, $n_{\text{bonus}}^i = 0$.*

Property 7. *Each bin with an uncolored item contains only that item.*

Proof. By line 3 of the Pack method, as soon as a bin becomes mixed, the colors of its items are fixed. By line 2 of the Pack method, an unmixed bin with an uncolored item does not receive a second item of the same type. □

In particular, no bin which contains an uncolored item is a mixed bin. The following important invariant generalizes a result for Super Harmonic (which is not formally proved in Seiden [98], but is quite easy to see for that algorithm).

Invariant 1. *If there exists an unmixed bin with red items of type j, then for any type i such that $\text{needs}(j) \leq \text{leaves}(i)$, there is no bin with a bonus item of type i and no unmixed bin with blue items of type i.*

Proof. As long as an unmixed red bin with items of some type j exists, no unmixed blue bin with items of type i for which $\text{needs}(j) \leq \text{leaves}(i)$ can be opened and vice versa (line 3 of Pack).

Now assume for a contradiction that there is an unmixed red bin with red items of type j (denote the first item in this bin by $\mathfrak{f}$) and a bin with a bonus item $\mathfrak{b}$ of type i. Assume $\mathfrak{b}$ arrived before $\mathfrak{f}$. Consider the point in time where $\mathfrak{f}$ arrived. After deciding that $\mathfrak{f}$ should be colored red in line 2 of Extreme Harmonic, we would have found that the second part of the condition in line 3 of Extreme Harmonic is true, and as a consequence would have made $\mathfrak{b}$ no longer be bonus, a contradiction to our assumption.

Now assume that $\mathfrak{f}$ arrived before $\mathfrak{b}$. In this case, either $\mathfrak{f}$ or the large item $\mathfrak{L}$ that is packed with $\mathfrak{b}$ arrived first. (Note that $\mathfrak{b}$ definitely arrived after $\mathfrak{L}$, or it would not have been made bonus.) Now $s(\mathfrak{L}) < 2/3$ since $\mathfrak{L}$ was packed with the medium item $\mathfrak{b}$. But $\text{redspace}_{\text{leaves}(t(\mathfrak{L}))} \geq 1/3$ by requirement R(7), $1/3 > \text{redspace}_{\text{leaves}(i)}$ by property 3, $\text{redspace}_{\text{leaves}(i)} \geq \text{redspace}_{\text{needs}(j)}$ by the assumption of the lemma. Thus $\text{redspace}_{\text{leaves}(t(\mathfrak{L}))} \geq \text{redspace}_{\text{needs}(j)}$, which implies $\text{leaves}(t(\mathfrak{L})) \geq \text{needs}(j)$, i.e., $\mathfrak{f}$ and $\mathfrak{L}$ are compatible according to definition 11. Hence, regardless of which item among these two arrived first, the algorithm does not pack them in different unmixed bins; the second arriving item would be packed at the latest by line 3 of Pack. □

4.2.4 Marking the items

Definition 12. *A* critical bin *for an Extreme Harmonic algorithm is a bin used in the optimal solution that contains a pair of items, one of a medium type j ($t_j \in (1/3, 1/2]$) and one of a large type i ($t_i \in (1/2, 2/3]$) such that $t_j + t_i > 1$ but $t_{j+1} + t_{i+1} < 1$.*

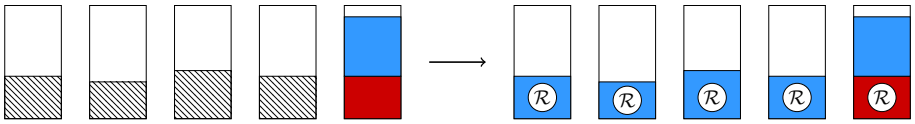
An example was given in fig. 4.1. By marking the medium items, we keep track of how many red and blue items of a given type j are in mixed bins. Blue medium items in mixed bins imply the existence of compatible small items in the input (which need to be packed somewhere in the optimal solution). Red medium items in mixed bins means that the algorithm managed to combine at least some pairs of medium and large items together into bins. In both cases, we have avoided the situation where the offline packing consists *only* of critical bins, whereas the online algorithm did not create *any* bins which contain a large and a medium item. We use three different marks, which together cover all the cases. Our marking is illustrated in fig. 4.6.

$\mathcal{R}$ For any medium type j, a fraction red_j of the items marked $\mathcal{R}$ are red, and all of these **red** items are packed into mixed bins (i.e., together with a large item).

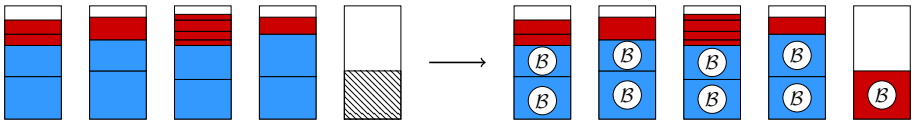
$\mathcal{B}$ For any medium type j, a fraction red_j of the items marked $\mathcal{B}$ are red, and the **blue** items are packed into mixed bins (i.e., together with small red items).

$\mathcal{N}$ For any medium type j, a fraction red_j of the items marked $\mathcal{N}$ are red, and **none** of the red and blue items marked $\mathcal{N}$ are packed into mixed bins.

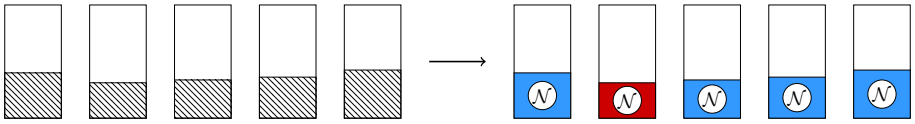
The algorithm Mark and Color is defined in algorithm 5. For a given type i and set $\mathcal{M} \in \{\mathcal{N}, \mathcal{B}, \mathcal{R}\}$, denote the number of red items by $n^i_r(\mathcal{M})$, and the total number of items by $n^i(\mathcal{M})$. Algorithm 5 is run every time after an item has been packed, and for every medium type i for which $\text{red}_i > 0$ separately. It divides the medium items into three sets $\mathcal{N}, \mathcal{B}$ and $\mathcal{R}$ (see fig. 4.6). Once assigned, an item remains in a set until the end of the input (after which it may be reassigned, see section 4.2.5). In many cases, the algorithm will have nothing to do, as none of the conditions hold. Therefore, some items will remain temporarily unmarked, in a set U. The set U does not contain the bonus items (in fact none of the sets does).



(a) Items get mark $\mathcal{R}$: uncolored items and a red item in a mixed bin. The bins with blue $\mathcal{R}$-items will receive an additional blue item of the same type before any new bin is opened for this type.



(b) Items get mark $\mathcal{B}$: a single uncolored item and blue items (in pairs) in mixed bins.



(c) Items get mark $\mathcal{N}$: a set of uncolored items. The bins with blue $\mathcal{N}$-items will receive an additional blue item of the same type before any new bin is opened for this type. See fig. 4.4.

Figure 4.6: Marking the items. For simplicity, we have taken $\text{red}_i = 1/9$ here (where i is the type of the medium items).

Algorithm 5 The algorithm Mark and Color as applied to *medium* items of type i for which $\text{red}_i > 0$.

if there is an unmarked blue item $\mathfrak{p}_1$ in a bin with a marked blue item $\mathfrak{p}_2$ **then**
  Give $\mathfrak{p}_1$ the same mark $\mathcal{M}$ as $\mathfrak{p}_2$.
  $n^i(\mathcal{M}) \leftarrow n^i(\mathcal{M}) + 1$
end if
Let $x_{\mathcal{R}}$ be the minimum integer value such that $\lfloor (n^i(\mathcal{R}) + 2x_{\mathcal{R}} + 1)\text{red}_i \rfloor > n^i_r(\mathcal{R})$.
if there exist $x_{\mathcal{R}}$ uncolored non-bonus items and one unmarked red or bonus item in a mixed bin **then**
  Mark these $x_{\mathcal{R}} + 1$ items $\mathcal{R}$. If there is a choice of items in mixed bins, use a bonus item if possible and color it red. Color the (other) uncolored items blue. If a bonus item is used, $n^i_{\text{bonus}} \leftarrow n^i_{\text{bonus}} - 1$.
  $n^i(\mathcal{R}) \leftarrow n^i(\mathcal{R}) + x_{\mathcal{R}} + 1$, $n^i_r(\mathcal{R}) \leftarrow n^i_r(\mathcal{R}) + 1$
end if
Let $x_{\mathcal{B}}$ be the minimum integer value such that $\lfloor (n^i(\mathcal{B}) + x_{\mathcal{B}} + 1)\text{red}_i \rfloor > n^i_r(\mathcal{B})$.
if there exists an uncolored non-bonus item and a set of mixed bins with two unmarked blue items each, which contains a number $x'_{\mathcal{B}} \in \{x_{\mathcal{B}}, x_{\mathcal{B}} + 1\}$ of blue items in total, **then**
  Mark these $x'_{\mathcal{B}}$ items $\mathcal{B}$ and color the uncolored item red.
  $n^i(\mathcal{B}) \leftarrow n^i(\mathcal{B}) + x'_{\mathcal{B}} + 1$, $n^i_r(\mathcal{B}) \leftarrow n^i_r(\mathcal{B}) + 1$
end if
Let $x_{\mathcal{N}}$ be the minimum integer value such that $\lfloor (n^i(\mathcal{N}) + 2x_{\mathcal{N}} + 1)\text{red}_i \rfloor > n^i_r(\mathcal{N})$.
if there exist $x_{\mathcal{N}} + 1$ uncolored non-bonus items **then**
  Mark the $x_{\mathcal{N}}$ largest uncolored items and the single smallest uncolored item $\mathfrak{p}$ with the mark $\mathcal{N}$. Color $\mathfrak{p}$ red and the other $x_{\mathcal{N}}$ items blue.
  $n^i(\mathcal{N}) \leftarrow n^i(\mathcal{N}) + x_{\mathcal{N}} + 1$, $n^i_r(\mathcal{N}) \leftarrow n^i_r(\mathcal{N}) + 1$
end if

Line 17 of Mark and Color ensures the following property, which was the point of postponing the coloring. Recall that early items are blue $\mathcal{N}$-items which did not get their color immediately and were packed one per bin (each late item is packed in a bin that already contains an early item).

Definition 13. *For some medium marked early blue item* $\mathfrak{p}$*, its* reference item*, denoted by* $r(\mathfrak{p})$*, is the red item that received its mark in the same iteration of Mark and Color. A set of bins whose (early) items received their marks in the same iteration of Mark and Color (i.e., a set of blue early items, their common reference item, and their corresponding late blue items) are called a* cluster*.*

Property 8. *Each early* $\mathcal{N}$*-item is at least as large as its reference item.*

After all items have arrived and after some post-processing, we will have

$$|n_r^i(\mathcal{M}) - n^i(\mathcal{M}) \cdot \mathrm{red}_i| = O(1) \text{ for } \mathcal{M} \in \{\mathcal{N}, \mathcal{B}, \mathcal{R}\} \text{ and each medium type } i \text{ with } \mathrm{red}_i > 0. \tag{4.1}$$

Each item will be marked according to the set to which it (initially) belongs. We will see that the values $x_{\mathcal{R}}, x_{\mathcal{B}}$ and $x_{\mathcal{N}}$ in Mark and Color are calculated in such a way that $n_r^i(\mathcal{M}) = \lfloor n^i(\mathcal{M}) \cdot \mathrm{red}_i \rfloor$ holds just before any assignment to $\mathcal{M} \in \{\mathcal{N}, \mathcal{B}, \mathcal{R}\}$. The proof is straightforward, but we need to be precise with the bound on $x_{\mathcal{N}}$ for later.

Note that Mark and Color never changes the values n_r^i and n^i. As we saw, the value n_r^i may be inaccurate for some types in any event. This will be fixed for small types in post-processing, whereas for medium types we will prove eq. (4.1). Of course, Mark and Color does change values $n_r^i(\mathcal{M})$ and $n^i(\mathcal{M})$ for $\mathcal{M} \in \{\mathcal{R}, \mathcal{N}, \mathcal{B}\}$ in order to record how many items with each mark there are (and these values *will* be accurate).

Lemma 1. *Let* $\mathcal{M} \in \{\mathcal{N}, \mathcal{R}\}$*. Just before assignments of new items to* $\mathcal{M}$ *in lines 7 to 8 or lines 17 to 18, for each medium type* i *such that* $\mathrm{red}_i > 0$*, we have* $n_r^i(\mathcal{M}) = \lfloor \mathrm{red}_i n^i(\mathcal{M}) \rfloor$ *and* $x_{\mathcal{M}} < 1/(2\mathrm{red}_i) + 1/2$*. Generally, we have* $n_r^i(\mathcal{M}) \in [\lfloor \mathrm{red}_i n^i(\mathcal{M}) \rfloor, \lfloor \mathrm{red}_i n^i(\mathcal{M}) \rfloor + 1]$.

Proof. Call the assignment of new items to $\mathcal{M}$ due to lines 7 to 8 or lines 17 to 18 *early assignments*.

At the beginning, we have $n_r^i(\mathcal{M}) = n^i(\mathcal{M}) = 0$. Thus the lemma holds at this time. When an early assignment takes place, $n^i(\mathcal{M})$ increases by $x_{\mathcal{M}} + 1$, and $n_r^i(\mathcal{M})$ by 1. By minimality of $x_{\mathcal{M}}$, just before any early assignment we have

$$\begin{aligned}
& \lfloor (n^i(\mathcal{M}) + 2(x_{\mathcal{M}} - 1) + 1)\mathrm{red}_i \rfloor \le n_r^i(\mathcal{M}) && (4.2)\\
\Longrightarrow\quad & (n^i(\mathcal{M}) + 2(x_{\mathcal{M}} - 1) + 1)\mathrm{red}_i < n_r^i(\mathcal{M}) + 1\\
\Longrightarrow\quad & (n^i(\mathcal{M}) + 2x_{\mathcal{M}} + 1)\mathrm{red}_i < n_r^i(\mathcal{M}) + 1 + 2\mathrm{red}_i\\
\Longrightarrow\quad & \lfloor (n^i(\mathcal{M}) + 2x_{\mathcal{M}} + 1)\mathrm{red}_i \rfloor < n_r^i(\mathcal{M}) + 1 + 2\mathrm{red}_i\\
\overset{R(4)}{\Longrightarrow}\quad & \lfloor (n^i(\mathcal{M}) + 2x_{\mathcal{M}} + 1)\mathrm{red}_i \rfloor - 1 < n_r^i(\mathcal{M}) + 1\\
\Longrightarrow\quad & \lfloor (n^i(\mathcal{M}) + 2x_{\mathcal{M}} + 1)\mathrm{red}_i \rfloor \le n_r^i(\mathcal{M}) + 1\\
\overset{\text{def. of } x_{\mathcal{M}}}{\Longrightarrow}\quad & \lfloor (n^i(\mathcal{M}) + 2x_{\mathcal{M}} + 1)\mathrm{red}_i \rfloor = n_r^i(\mathcal{M}) + 1
\end{aligned}$$

This immediately implies that right after an early assignment to $\mathcal{M}$,

$$\lfloor (n^i(\mathcal{M}) + x_{\mathcal{M}})\mathrm{red}_i \rfloor = n_r^i(\mathcal{M}). \tag{4.3}$$

There are then $x_{\mathcal{M}}$ bins with one early blue medium item of type i. Extreme Harmonic will put the next arriving blue items of this type into these $x_{\mathcal{M}}$ bins (one additional item per bin) before opening any new bins. All of these late blue items are assigned to $\mathcal{M}$ and $n^i(\mathcal{M})$ is increased accordingly in lines 2 to 3, so eventually $n_r^i(\mathcal{M}) = \lfloor \mathrm{red}_i n^i(\mathcal{M}) \rfloor$.

After that, $n^i(\mathcal{M})$ and $n_r^i(\mathcal{M})$ remain unchanged until the next early assignment of items to $\mathcal{M}$. Hence before an early assignment of items to $\mathcal{M}$, the first claimed equality holds.

This equality together with ineq. (4.2) gives

$$\lfloor n^i(\mathcal{M})\mathrm{red}_i + (2(x_{\mathcal{M}} - 1) + 1)\mathrm{red}_i \rfloor \leq \lfloor n^i(\mathcal{M})\mathrm{red}_i \rfloor$$

which implies $(2(x_{\mathcal{M}} - 1) + 1)\mathrm{red}_i < 1$ and thus $x_{\mathcal{M}} < 1/(2\mathrm{red}_i) + 1/2 < 1/\mathrm{red}_i$. This, together with eq. (4.3), implies $n_r^i(\mathcal{M}) < \lfloor (n^i(\mathcal{M}) \rfloor + 1$ (note that $n_r^i(\mathcal{M})$ is largest relative to $n^i(\mathcal{M})$ right after an early assignment to $\mathcal{M}$, i.e., when eq. (4.3) holds). □

Corollary 1. *After each execution of Mark and Color and for each medium type i such that* $\mathrm{red}_i > 0$, $n^i(\mathsf{U}) \leq 1/\mathrm{red}_i$.

Proof. We have $x_{\mathcal{N}} < 1/\mathrm{red}_i$ and $x_{\mathcal{R}} < 1/\mathrm{red}_i$ by lemma 1, so at the latest when $1/\mathrm{red}_i + 1$ uncolored non-bonus items exist, they are marked and colored. □

Lemma 2. *At all times and for each medium type i such that* $\mathrm{red}_i > 0$, $n_r^i(\mathcal{B}) = \lfloor \mathrm{red}_i n^i(\mathcal{B}) \rfloor$ *and* $x_{\mathcal{B}} < 1/\mathrm{red}_i$.

Proof. We use similar calculations to the proof of lemma 1. At the beginning, all counters are zero. When Mark and Color is about to assign items to $\mathcal{B}$, we have $\lfloor \mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}} + 1) \rfloor = n_r^i(\mathcal{B}) + 1$: Assume the contrary, that is, $\lfloor \mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}} + 1) \rfloor \geq n_r^i(\mathcal{B}) + 2$. Then $\lfloor \mathrm{red}_i(n^i(\mathcal{B}) + (x_{\mathcal{B}} - 1) + 1) \rfloor \geq n_r^i(\mathcal{B}) + 1 > n_r^i(\mathcal{B})$, a contradiction to minimality of $x_{\mathcal{B}}$. We also have $\lfloor \mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}} + 2) \rfloor = n_r^i(\mathcal{B}) + 1$. If $\mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}} + 1)$ is integral, this is clear by requirement R(4). Otherwise, assuming $\lfloor \mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}} + 2) \rfloor = n_r^i(\mathcal{B}) + 2$ would mean that $\lceil \mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}} + 1) \rceil - \mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}} + 1) < 1/3$, hence $\mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}} + 1) - \lfloor \mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}} + 1) \rfloor > 2/3$, and hence by requirement R(4) $\mathrm{red}_i(n^i(\mathcal{B}) + (x_{\mathcal{B}} - 1) + 1) > \lfloor \mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}} + 1) \rfloor$, again contradicting minimality of $x_{\mathcal{B}}$. Having $\lfloor \mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}} + 1) \rfloor = \lfloor \mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}} + 2) \rfloor = n_r^i(\mathcal{B}) + 1$ immediately implies that after each assignment, we have $\lfloor \mathrm{red}_i n^i(\mathcal{B}) \rfloor = n_r^i(\mathcal{B})$. By minimality of $x_{\mathcal{B}}$, we also conclude $\lfloor \mathrm{red}_i(n^i(\mathcal{B}) + x_{\mathcal{B}}) \rfloor = n_r^i(\mathcal{B})$, so $x_{\mathcal{B}} \cdot \mathrm{red}_i < 1$. □

4.2.5 Post-processing

Since we consider only the asymptotic competitive ratio in this paper, it is sufficient to prove that a certain ratio holds for all but a constant number of bins: such bins are counted in the additive constant. We will perform a sequence (of constant length) of removals of bins in this section. We will also change the marks of some items to better reflect the actual output, fix the type and color of any remaining bonus items and reduce the sizes of some items to match the values used by Extreme Harmonic in its accounting (see line 11 of algorithm 3). We will now establish – using such modifications – some *packing properties*. After establishing such a property, we will

argue that it will not be affected by further modifications. In the end, all packing properties will hold and build the basis for our anaylsis.

Lemmas 1 and 2 give us the first packing property.

Packing Property 1. *For $\mathcal{M} \in \{\mathcal{N}, \mathcal{R}, \mathcal{B}\}$, we have $\left|n^i(\mathcal{M}) \cdot \mathrm{red}_i - n^i_r(\mathcal{M})\right| = O(1)$.*

The next packing property follows from the way Mark and Color selects the items to mark.

Packing Property 2. *All bins with blue $\mathcal{B}$-items or red $\mathcal{R}$-items are mixed.*

We remove all bins with unmarked items (but not the bonus items); according to corollary 1, there are at most $\sum_{i:\mathrm{red}_i>0} 1/red_i$ such bins. This establishes the following packing property:

Packing Property 3. *All medium non-bonus items are marked.*

We remove any bins with a *single* blue $\mathcal{N}$- or $\mathcal{R}$-item $\mathfrak{p}$, as well as all bins in the same cluster as $\mathfrak{p}$. Line 7 or line 17 of Mark and Color are only executed if all blue items that were assigned to $\mathcal{N}$ or $\mathcal{R}$ in a previous run of Mark and Color are already packed into bins with two blue items, since the algorithm Pack prefers to pack a new blue item into an existing blue-open bin. Thus, when we remove all bins with single $\mathcal{N}$- or $\mathcal{R}$-items, this is at most $\sum_{i:\mathrm{red}_i>0}(1+1/\mathrm{red}_i)$ bins by lemma 1. This does also not invalidate packing property 1, following the argumentation from the proof of lemma 1. We established the following packing property:

Packing Property 4. *Each blue item in $\mathcal{N}, \mathcal{R}$ and $\mathcal{B}$ is packed in a bin that contains two blue items.*

Finally, we remove all open bins, which are at most $O(N^2)$ by property 4. We also remove the single bin with items of type N of total size at most $1-\varepsilon$, if it exists (see property 1). We thus get the next two packing properties.

Packing Property 5. *No open bins exist. That is, every bin that contains red items of type i contains redfit_i such items, and every bin that contains blue items of type i contains $\mathrm{bluefit}_i$ such items, for every type $i < N$.*

Packing Property 6. *All bins with items of type N are at least $1-\varepsilon$ full.*

Final marking An overview of our changes of marks and sizes is given in fig. 4.7. We will change marks of some items to $\mathcal{R}$ or $\mathcal{B}$ if such marks are appropriate. To do this, we run algorithm 6 for every medium type i separately. Note that seemingly wrongly marked items like the ones we look for in algorithm 6 can indeed exist because while the algorithm is running we only mark each item once, when it is assigned to a set; other items could arrive later and be packed with it, invalidating its mark.

Lemma 3. *Algorithm 6 does not affect packing properties 1 to 6.*

Proof. Packing properties 3 to 6 are obviously not affected by changing marks. Packing property 2 is maintained by the way we select items that are moved to other sets. Packing property 1 is maintained because we change marks in the correct proportions:

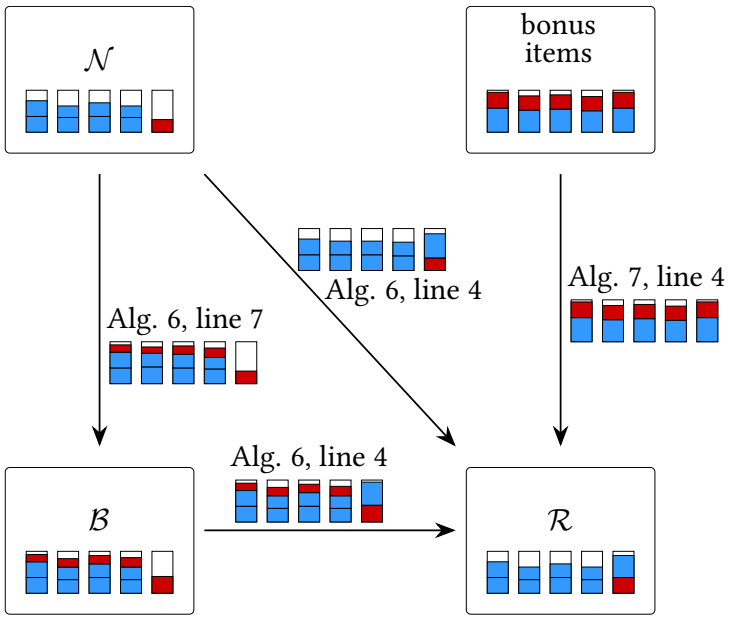


Figure 4.7: Reassigning marks after the input is complete and changing some items to get rid of bonus items. Items are sorted into their correct sets whenever possible, updating the marks that they received while the algorithm was running. Some item sizes are reduced (!). The bins next to the arrows indicate what sets of bins are being reassigned. Note that, when shifting items from $\mathcal{B}$ to $\mathcal{R}$, we obtain bins with two blue $\mathcal{R}$-items *and some red items*. This is no problem of course, as we do not require that the blue $\mathcal{R}$-item are in unmixed bins, but it is not depicted here.

If $\left|n^i(\mathcal{M}) \cdot \mathrm{red}_i - n_r^i(\mathcal{M})\right| \leq c$ holds for some $\mathcal{M} \in \{\mathcal{N}, \mathcal{R}, \mathcal{B}\}$ and constant c before shifting items to this set, then afterwards we have

$$\begin{aligned} n_r^i(\mathcal{M}) + \lfloor \mathrm{red}_i T \rfloor &\in \Big[\lfloor \mathrm{red}_i n^i(\mathcal{M}) \rfloor - c + \lfloor \mathrm{red}_i T \rfloor, \lfloor \mathrm{red}_i n^i(\mathcal{M}) \rfloor + c + \lfloor \mathrm{red}_i T \rfloor\Big] \\ &\subseteq \Big[\lfloor \mathrm{red}_i (n^i(\mathcal{M}) + T) \rfloor - O(c), \lfloor \mathrm{red}_i (n^i(\mathcal{M}) + T) \rfloor + O(c)\Big]. \end{aligned}$$

Similarly, when shifting items from $\mathcal{M}$ to some other set, we get

$$\begin{aligned} n_r^i(\mathcal{M}) - \lfloor \mathrm{red}_i T \rfloor &\in \Big[\lfloor \mathrm{red}_i n^i(\mathcal{M}) \rfloor - c - \lfloor \mathrm{red}_i T \rfloor, \lfloor \mathrm{red}_i n^i(\mathcal{M}) \rfloor + c - \lfloor \mathrm{red}_i T \rfloor\Big] \\ &\subseteq \Big[\lfloor \mathrm{red}_i (n^i(\mathcal{M}) - T) \rfloor - O(c), \lfloor \mathrm{red}_i (n^i(\mathcal{M}) - T) \rfloor + O(c)\Big]. \end{aligned}$$

As we only do this shifting a constant number of times, packing property 1 is maintained. □

Instead of the process described in algorithm 6, an easier approach might seem to be the following. For changing marks from $\mathcal{N}$ to $\mathcal{R}$, we could simply take the clusters of the red $\mathcal{N}$-items in the mixed bins. The problem with this approach is that not all these clusters have the same size in general, since $x_{\mathcal{N}}$ may vary. This means the ratio red_i would possibly not be maintained for $\mathcal{R}$ (and then also not for $\mathcal{N}$).

Packing Property 7. *No bins with items in $\mathcal{N}$ are mixed. No bins with* red *items in $\mathcal{B}$ are mixed.*

Algorithm 6 Final marking for items of type i in Extreme Harmonic algorithms. Again we only consider items of medium type i.

1: Sort the bins with two blue $\mathcal{N}$-items in order of increasing size of the early $\mathcal{N}$-items in these bins.
2: **for** $\mathcal{M} = \{\mathcal{N}, \mathcal{B}\}$ **do**
3: Let T be the largest integer value such that there exist
- $\lfloor \mathrm{red}_i T \rfloor$ red $\mathcal{M}$-items in mixed bins (one per bin) and
- $(T - \lfloor \mathrm{red}_i T \rfloor)/2$ bins with (two) blue $\mathcal{M}$-items (so $(T - \lfloor \mathrm{red}_i T \rfloor)/2 \in \mathbb{N}$)

4: Assign the $\lfloor \mathrm{red}_i T \rfloor$ largest red $\mathcal{M}$-items in mixed bins and the blue $\mathcal{M}$-items in the first $(T - \lfloor \mathrm{red}_i T \rfloor)/2$ bins in the sorted order to $\mathcal{R}$
5: **end for**
6: Let T be the largest integer value such that there exist
- $\lfloor \mathrm{red}_i T \rfloor$ red $\mathcal{N}$-items
- $(T - \lfloor \mathrm{red}_i T \rfloor)/2$ mixed bins with (two) blue $\mathcal{N}$-items

7: Assign the $\lfloor \mathrm{red}_i T \rfloor$ largest red $\mathcal{N}$-items and the blue $\mathcal{N}$-items in the first $(T - \lfloor \mathrm{red}_i T \rfloor)/2$ mixed bins in the sorted order that were not yet reassigned to $\mathcal{R}$, to $\mathcal{B}$

Lemma 4. *After running algorithm 6, only constantly many bins need to be removed in order to ensure that packing property 7 holds. Packing properties 1 to 6 are maintained.*

Proof. Let us fix a medium type i. After the first loop is finished, there can be at most constantly many red $\mathcal{N}$-items and $\mathcal{B}$-items in mixed bins, since these sets of items are both colored with the correct proportion of red items by packing property 1 and we move a maximal subset of items with the correct proportion to $\mathcal{R}$. After algorithm 6 completes, there are at most constantly many blue $\mathcal{N}$-items in mixed bins for the same reason. We can remove all of these bins at the end if needed. This does not affect any of packing properties 1 to 6. □

The following lemma helps us to bound the optimal solution later.

Lemma 5. *Let the smallest medium red item of type i in $\mathcal{N}$ be $\mathfrak{r}_i$. It is packed alone in a bin. At most $\frac{1-\mathrm{red}_i}{2}(n^i(\mathcal{R}) + n^i(\mathcal{N})) + O(1)$ items in $\mathcal{N}$ have size less than $s(\mathfrak{r}_i)$.*

Proof. Item $\mathfrak{r}_i$ is packed alone by packing property 7 and the fact that $\mathrm{redfit}_i = 1$ for medium type i. Each red $\mathcal{N}$-item of type i has size at least $s(\mathfrak{r}_i)$ by definition of $\mathfrak{r}_i$. Furthermore, each *early* blue $\mathcal{N}$-item of type i has size at least $s(r(\mathfrak{p}))$, where $r(\mathfrak{p})$ is the reference item of $\mathfrak{p}$ (property 8). However, it is possible that the bin containing $r(\mathfrak{p})$ received an additional (large, blue) item later. In that case, after post-processing, the item $r(\mathfrak{p})$ does not have mark $\mathcal{N}$ anymore, so it is not considered when determining $\mathfrak{r}_i$, and may in fact be smaller than $\mathfrak{r}_i$; thus also $\mathfrak{p}$ may be smaller than $\mathfrak{r}_i$ although it is an early blue $\mathcal{N}$-item. In algorithm 6, we therefore take care to always select the bins with the smallest early blue $\mathcal{N}$-items (line 1).

Thus, there are two kinds of items in $\mathcal{N}$ that can be smaller than $\mathfrak{r}_i$: Late blue $\mathcal{N}$-items and early blue $\mathcal{N}$-items whose reference items are shifted to $\mathcal{R}$ by algorithm 6.

We first give an upper bound for the number of items of the second kind. Let $z = \lfloor \mathrm{red}_i T \rfloor$ be the number of red $\mathcal{N}$-items in mixed bins that receive the mark $\mathcal{R}$ in line 4 of algorithm 6. Then the total number of early items that are in the same

clusters as these z items is upper bounded by $z/(2\text{red}_i)+z/2$ by lemma 1. We transfer in total $(T-z)/2$ early items from $\mathcal{N}$ to $\mathcal{R}$ and these are the smallest early items. The number of early items that do not get transferred and are potentially smaller than $\mathfrak{r}_i$ is therefore at most $z/(2\text{red}_i)+z-T/2 \leq z$ since $T \geq z/\text{red}_i$. Clearly, since we move z red $\mathcal{N}$-items to $\mathcal{R}$, z is at most $\text{red}_i n^i(\mathcal{R})$ afterwards.

Next, we give an upper bound for the first kind of items that are potentially smaller than $\mathfrak{r}_i$, i.e., the late blue $\mathcal{N}$-items. There are $n^i(\mathcal{N})-n_r^i(\mathcal{N}) = n^i(\mathcal{N})-\lfloor \text{red}_i n^i(\mathcal{N})\rfloor \pm O(1)$ blue items in $\mathcal{N}$ (using packing property 1). Half of them are packed as late items. We have $\frac{1}{2}(n^i(\mathcal{N})-\lfloor \text{red}_i n^i(\mathcal{N})\rfloor \pm O(1)) \leq \frac{1-\text{red}_i}{2} n^i(\mathcal{N}) + \frac{1}{2} + O(1)$.

Since $\text{red}_i < \frac{1-\text{red}_i}{2}$ by requirement R(4), the lemma follows. □

A modification of the input In line 20 of Extreme Harmonic, bonus items are created. These are medium items which are packed as red items (each such item is in a bin with a large blue item) but violate the ratio red_i. Some of them may still be bonus when the algorithm has finished. Also, some of them may be labeled with a different type than the type they belong to according to their size. We call such items *reduced* items. Note that Extreme Harmonic treated each reduced item as small red items in its accounting (but had in fact packed the larger bonus item). All reduced items are in mixed bins. They are not counted as bonus items.

Algorithm 7 Modifying the input after packing all items

1: Let the number of bonus items of type i be T. // *These are not reduced items*
2: Color $\lfloor \frac{2\text{red}_i T}{1+\text{red}_i} \rfloor$ of these items red and the others blue. Mark them all $\mathcal{R}$.
3: Reduce the size of blue large items in the bins with (now) blue medium items of type i to t_i.
4: Mark all of these items $\mathcal{R}$ as well.
5: $n^i(\mathcal{R}) \leftarrow n^i(\mathcal{R}) + 2T - \lfloor \frac{2\text{red}_i T}{1+\text{red}_i} \rfloor$.
6: $n_r^i(\mathcal{R}) \leftarrow n_r^i(\mathcal{R}) + \lfloor \frac{2\text{red}_i T}{1+\text{red}_i} \rfloor$.
7: **for each** reduced item $\mathfrak{p}$ **do**
8: Let j be the type with which $\mathfrak{p}$ is labeled.
9: Split up $\mathfrak{p}$ into redfit_j red items of size $s(\mathfrak{p})/\text{redfit}_j$.
10: Reduce the size of the newly created items until they belong to type j.
11: **end for**

After Extreme Harmonic has finished, and the steps previously described in this sections have been applied, we modify the packing that it outputs as described in algorithm 7. Again we run this algorithm for every medium type i. The post-processing is illustrated in fig. 4.5; the process in lines 2 to 6 is illustrated in fig. 4.5c, the process in lines 8 to 10 in fig. 4.5b.

Lemma 6. *Denote the set of items in a given packing P by σ. Denote the set of items after applying algorithm 7 to the packing P by σ'. Then P induces a valid packing for σ', and $\text{OPT}(\sigma') \leq \text{OPT}(\sigma)$.*

Proof. In line 3 of algorithm 7, items are only made smaller. In line 9, a medium item of type i is split into redfit_j items of some type j. The condition for an item to be labeled with type j in line 3 of Extreme Harmonic is that j is a small type.

By definition of redfit_j and $\text{redspace}_{\text{needs}(j)}$, we have that redfit_j items of type j have total size at most $\text{redspace}_{\text{needs}(j)}$. Since j is a small type, this value is less

than 1/3 by requirement R(1). This means the newly created items occupy less space than the medium item that they replace. Hence, in both cases we do not increase the amount of occupied space in any bin.

The inequality follows by choosing P to be an optimal packing for σ. □

Lemma 7. *Lemma 5 still holds after executing algorithm 7.*

Proof. Algorithm 7 creates new items only with mark $\mathcal{R}$. Therefore, the number of "problematic items" that we want to upper bound, that is, the number of $\mathcal{N}$-items of size less than $s(\mathfrak{r}_i)$, does not increase. As we only increase $n^i(\mathcal{R})$ in algorithm 7, the upper bound in lemma 5 is not decreased. □

Theorem 1. *For a given input σ, denote the result of all the post-processing done in this section by $\sigma' = \{\mathfrak{p}_1, \dots, \mathfrak{p}_n\}$. Packing properties 1 to 7 and invariant 1 still hold after post-processing. For any type i, at the end we have $|n^i_r - \mathrm{red}_i n^i| = O(1)$, where n^i_r counts the (correct) total number of red items of type i after postprocessing. There are no bonus items, and the optimal cost to pack the input did not increase in post-processing.*

Proof. Let P_0 be the packing of σ that is output by $\mathcal{A}$. Let P_1 be the packing after running algorithm 6, and let the items packed into P_1 be σ_1. By lemma 4, the packing properties hold before algorithm 7 is executed. Packing property 1 is not affected by lines 2 to 6, as the additional items marked $\mathcal{R}$ are colored with the correct ratio: We have $r = \lfloor \frac{2\mathrm{red}_i T}{1+\mathrm{red}_i} \rfloor$ red items, $b = 2(T-r) = 2T - 2\lfloor \frac{2\mathrm{red}_i T}{1+\mathrm{red}_i} \rfloor$ blue items, that means $r + b = 2T - \lfloor \frac{2\mathrm{red}_i T}{1+\mathrm{red}_i} \rfloor$ items in total in $r + b/2 = T$ bins, and $|\mathrm{red}_i(r+b) - r| = O(1)$. Packing property 2 obviously holds as the new red $\mathcal{R}$-items are in mixed bins as they were bonus items before. Packing property 4 holds as in the bins with new blue $\mathcal{R}$-items, the formerly large items are shrunk to become $\mathcal{R}$-items of the same medium type as well. Packing properties 3 and 5 to 7 are not affected.

Consider some medium type i. Invariant 1 is not affected by any change of marks or removal of bins. The effect of lines 2 to 6 of algorithm 7 is that some bins with bonus items of type i are replaced with unmixed bins with blue type i items. As invariant 1 held before where we had bonus items of type i, we know that there is no unmixed bin with red items of type j such that $\mathrm{needs}(j) \leq \mathrm{leaves}(i)$, so the invariant still holds after we created the new unmixed bins with blue type i items.

To get from P_0 to P_1, we only removed some bins (and changed marks, which are irrelevant for the optimal solution). Hence $\textsc{opt}(\sigma_1) \leq \textsc{opt}(\sigma)$. We can thus apply lemma 6 to the optimal packing for σ_1 to get the last claim. All bonus items are removed by algorithm 7.

It remains to argue about the correct proportion of red to blue items for all types. For medium types, by packing property 1 the ratios are correct within each set $\mathcal{R}, \mathcal{B}, \mathcal{N}$, so they are also correct in total. For a small type i, we have $|n^i_r - \mathrm{red}_i n^i| = O(1)$ by property 5. The only effect of post-processing is that afterwards, n^i_r counts the actual number of red items of type i in σ_1. (Some of these red items replace bonus items in σ_1, but the algorithm already counted them in the value n^i_r.) □

4.2.6 Weighting functions

Let $\mathcal{A}$ be an Extreme Harmonic algorithm. For analyzing the competitive ratio of $\mathcal{A}$, we will use the well-known technique of weighting functions. The idea of this technique is the following. We assign weights to each item such that the number of

bins that our algorithm uses in order to pack a specific input is equal (up to an additive constant) to the sum of the weights of all items in this input. Then, we determine the average weight that can be packed in a bin in the optimal solution. This average weight for a single bin gives us an upper bound on the competitive ratio. In order to use this technique, we now define a set of weighting functions.

Fix an input sequence σ. Denote the result of post-processing σ by $\sigma' = \{\mathfrak{p}_1, \ldots, \mathfrak{p}_n\}$. Let P be the packing of σ that is output by $\mathcal{A}$. Let P' be the packing of σ' induced by P (lemma 6).

From this point on, our analysis is purely based on the structural properties of the packing P' that we established in theorem 1. We view σ' only as a set of items and not as a list. We prove in theorem 2 below that this is justified. In particular, we do *not* make any statement about $\mathcal{A}(\sigma')$, since the post-processing done in algorithm 6 means that some items (e.g., the ones introduced in lines 8 to 10) do not have clearly defined arrival times, and it is not obvious how to define arrival times for them in order to ensure that $\mathcal{A}(\sigma') = \mathcal{A}(\sigma)$. Recall that the class of an item of type t is $\text{leaves}(t)$, if it is blue, and $\text{needs}(t)$ if it is red.

Lemma 8. *For $k \in \{1, \ldots, K\}$, red items of class k are either all medium or all small. If they are medium, they are of the unique type t such that $k = \text{needs}(t)$.*

Proof. If for a red item of type t we have $\text{redspace}_{\text{needs}(t)} > 1/3$, then it is a medium item by requirement R(1); in this case, type t is the only type such that $\text{needs}(t) = k$ since each medium type is in a different class by requirement R(3). For each small item $\mathfrak{p}$ we have $\text{redspace}_{\text{needs}(t(\mathfrak{p}))} \leq 1/3$, i.e., no class can contain items of small *and* medium types. □

The class of a bin with red items is the class of those red items. This is well-defined, as each bin contains red items of only one type.

Definition 14. *Let k be the minimum class of any unmixed red bin. Let $\mathfrak{r}$ be a smallest item in the unmixed red bins of class k. If all red items are in mixed bins, we define $k = K + 1$ (and $\mathfrak{r}$ is left undefined).*

If $k \in \{1, \ldots, K\}$, then by this definition we have $k = \text{needs}(t(\mathfrak{r}))$. If redspace_k is at most $1/3$, there may be several red items in one bin, as in Super Harmonic (in Harmonic++, there is always at most one red item per bin).[7] Also, there can be several types t such that $k = \text{needs}(t)$.

We follow Seiden's proof, adapting it to take the marks into account. In order to define the weight functions, it is convenient to introduce some additional types. Note that the algorithm does not depend on the weight functions in any way. It is also unaware of the added type thresholds. First of all, for each i such that $1/3 < \text{redspace}_i < 1/2$, we add a threshold $1 - \text{redspace}_i$ between $t_2 = 2/3$ and $t_3 = 1/2$ (see requirement R(5)). For a type t with upper bound $1 - \text{redspace}_i$ we define $\text{leaves}(t) = i$. We furthermore add a threshold $1 - s(\mathfrak{r})$ in case $\mathfrak{r}$ is medium. This splits an existing type into two types. For the new type t^1 with upper bound $1 - s(\mathfrak{r})$, we

[7] At this point we note that there is a minor inaccuracy in the corresponding proof in Seiden [98]. He defines an item e as the smallest red item in an indeterminate red group bin, and proceeds to argue using the class of e (where we use k). However, that class does not need to be monotone in the size of e, so there could be a larger red item of a smaller class that is in an indeterminate group bin! Instead, e should be defined based on the smallest class of an unmixed red bin, as we do above. The proof as written by Seiden only works for algorithms like Harmonic++ which pack only one red item per bin.

define leaves(t^1) = k, where k = needs($t(\mathfrak{r})$). For the new type t^2 with lower bound $1-s(\mathfrak{r})$, we define leaves(t^2) = $k-1$. We see that the function leaves now depends on the size of the item $\mathfrak{r}$ and thus we will write $\text{leaves}_{s(\mathfrak{r})}$. To maintain consistency with the rest of the chapter, we add negative indices for the types to maintain $t_3 = 1/2$. That is, if there are a values in Redspace in the range $(1/3, 1/2)$, the corresponding values $1-\text{redspace}_i$ and the threshold $1-s(\mathfrak{r})$ (if $\mathfrak{r}$ is medium) are stored in ascending order in the values $t_2, t_1, \ldots, t_{2-a}$, and $t_{1-a} = 2/3$, $t_{-a} = 1$.

For large items, the value of the function leaves(i) is only used by the algorithm to check whether *small* items can be combined with them. Moreover, for small items, the only relevant piece of information is that at least 1/3 of space is left by large items. An Extreme Harmonic algorithm defines leaves(2) such that $\text{redspace}_{\text{leaves}(2)} = 1/3$ (and then ignores this value when considering to pack a medium item with a large item). The additional types simply make the function leaves more accurate, in particular with the threshold $1 - s(\mathfrak{r})$, which the algorithm does not know. It can be seen that the definition of k (and $\mathfrak{r}$) is not affected by these new types, as only types of large (i.e., blue) items are changed, and k and $\mathfrak{r}$ are defined based on unmixed red bins.

The weights of an item $\mathfrak{p}$ will depend on $s(\mathfrak{r})$, the class of the red and blue items of type $t(\mathfrak{p})$ relative to k, and the mark of $\mathfrak{p}$. This means we essentially define them for every possible input sequence separately. The value of k and $s(\mathfrak{r})$ (and the marks) become clear by running the algorithm. We do not write the dependence on σ explicitly since we have fixed σ in this section.

The two weight functions of an item of size x, type t and mark $\mathcal{M}$ are given below. For convenience, define $\beta_t = \frac{1-\text{red}_t}{\text{bluefit}_t}$ and $\rho_t = \frac{\text{red}_t}{\text{redfit}_t}$; these represent the "blue weight" and "red weight" of type t in the sense that if we have m items of type t, the red items are packed in $\rho_t m$ bins and the blue items are packed in $\beta_t m$ bins. That way (and using packing property 5), w_k counts all bins with blue items (and some additional bins with red items), and $v_{k,s(\mathfrak{r})}$ counts all bins with red items (and some additional bins with blue items). We will later show in theorem 2 that both functions indeed count all bins in the packing produced by the algorithm.

The weighting function w only depends on the size, type and mark of the item $\mathfrak{p}$ and on k:

$$w_k(\mathfrak{p}) = w_k(x,t,\mathcal{M}) = \begin{cases} \beta_t + \rho_t & \text{if } t < N \text{ and } (\text{needs}(t) > k \text{ or} \\ & \quad \text{needs}(t) = 0 \text{ or} \\ & \quad (\text{needs}(t) = k \text{ and } \mathcal{M} \neq \mathcal{R})) \\ \beta_t & \text{if } t < N \text{ and either } 0 < \text{needs}(t) < k \\ & \text{or needs}(t) = k, \mathcal{M} = \mathcal{R} \\ \frac{1}{1-\varepsilon} x & \text{if } t = N \end{cases}$$

Recall that $\varepsilon = t_N$. Non-medium items have no mark and are handled under the case $\mathcal{M} \neq R$. (Unmarked *medium* items were removed in the previous section). By definition of k and packing property 7, we have $\mathcal{M} = \mathcal{R}$ for all items with type t such that needs(t) < k. For simplicity, we ignore the markings for any type t with needs(t) > k, essentially assuming that there are no items of such types that are marked $\mathcal{R}$. It is clear that this assumption can only increase the weight of any item. Note that $w_k(\mathfrak{p})$ does not depend on $s(\mathfrak{r})$ or the added types, as $\text{red}_t = 0$ and needs(t) = 0 for all items larger than 1/2.

In contrast, our second weighting function v depends on $\text{leaves}_{s(\mathfrak{r})}(t)$ and thus also on the threshold $1 - s(\mathfrak{r})$ if $\mathfrak{r}$ is medium as described above. Therefore, v depends on the size and type of the item $\mathfrak{p}$ and on k and the size of $\mathfrak{r}$:

$$v_{k,s(\mathfrak{r})}(\mathfrak{p}) = v_{k,s(\mathfrak{r})}(x,t) = \begin{cases} \beta_t + \rho_t & \text{if } t < N, \text{leaves}_{s(\mathfrak{r})}(t) < k \\ \rho_t & \text{if } t < N, \text{leaves}_{s(\mathfrak{r})}(t) \geq k \\ \frac{1}{1-\varepsilon} x & \text{if } t = N \end{cases}$$

Note that for any item $\mathfrak{p}$, we have $\tilde{v}_k(\mathfrak{p}) \geq \tilde{v}_{k,s(\mathfrak{r})}(\mathfrak{p})$ since $1 - s(\mathfrak{r}) \geq 1 - t_{t(\mathfrak{r})}$, and this is the point at which the $\text{leaves}_{s(\mathfrak{r})}$ function drops below k. Thus, we define $v_k(\mathfrak{p}) = v_{k,t_{t(\mathfrak{r})}}(\mathfrak{p})$.

Theorem 2. *For any input σ and Extreme Harmonic algorithm $\mathcal{A}$, defining k as above*[8] *we have*

$$\mathcal{A}(\sigma) \leq \min\left\{ \sum_{i=1}^{n} w_k(\mathfrak{p}_i), \sum_{i=1}^{n} v_k(\mathfrak{p}_i) \right\} + O(1) \tag{4.4}$$

Proof. Our goal is to upper bound $\mathcal{A}(\sigma)$ by the weights of the items $\mathfrak{p}_1, \ldots, \mathfrak{p}_n$, which are the items in σ'. We will show that the number of bins in the packing P' is upper bounded by the min-term in ineq. (4.4), with the additive constant $O(1)$ corresponding to the bins removed in post-processing. We follow the line of the corresponding proof in Seiden [98].

Let Tiny be the total size of the items of type N in σ'. Let UnmixedRed be the number of unmixed red bins in P'. Let B_i and R_i be the number of bins in P' containing blue items of class i and type less than N, and red items of class i, respectively. Note that this means that mixed bins are counted twice.

If UnmixedRed $= 0$, every red item is placed in a bin with one or more blue items, and $k = K + 1$. In this case, the total number of bins in P' is exactly the total number of bins containing blue items. Each bin containing items of type N contains at least a total size of $1 - \varepsilon$ due to packing property 6. The bins used to pack Tiny are pure blue and $\sum_{t(\mathfrak{p}_i)=N} w_{K+1}(\mathfrak{p}_i) = \sum_{t(\mathfrak{p}_i)=N} v_{K+1}(\mathfrak{p}_i) \geq \text{Tiny}/(1-\varepsilon)$. For each item $\mathfrak{p}$ of type $t < N$, we have $w_{K+1}(\mathfrak{p}) = \beta_t = \frac{1-\text{red}_t}{\text{bluefit}_t} \leq v_{K+1}(\mathfrak{p})$. We see that w_{K+1} counts all the bins with blue items (see packing property 5), and

$$\mathcal{A}(\sigma) \leq \frac{\text{Tiny}}{1-\varepsilon} + \sum_{i=0}^{K} B_i \leq \sum_{i=1}^{n} w_{K+1}(\mathfrak{p}_i)$$

(since B_0 does not include bins with items of type N).

If UnmixedRed > 0, then $k = \text{needs}(t(\mathfrak{r}))$, and there is an unmixed red bin of class k. By invariant 1, all bins with a blue item of class $i \geq k$ must be *mixed* bins. These are the bins which contain blue items of any type j such that $\text{leaves}(j) \geq k$; if $\mathfrak{r}$ is medium, this means exactly the large items with size at most $1 - s(\mathfrak{r})$. We conclude

$$\text{UnmixedRed} \leq \sum_{i=1}^{K} R_i - \sum_{i=k}^{K} B_i. \tag{4.5}$$

[8] Seiden expresses the upper bound as a maximum over k, even though for a fixed input sequence, the value of k is fixed. While the resulting expression is correct, we prefer this formulation.

Let $R_k(-\mathcal{R})$ be the number of bins in P' containing red items of class k that are not marked $\mathcal{R}$. If items of class k are not medium, then $R_k(-\mathcal{R}) = R_k$. This is a well-defined distinction by lemma 8. Let R_i^* be the number of *unmixed* bins in P' containing red items of class i. Since every red item with class less than k (that is, red items of any type j such that $\text{needs}(j) < k$) is placed in a mixed bin by definition of k, we have

$$\textsc{UnmixedRed} \le \sum_{i=k+1}^{K} R_i^* + R_k(-\mathcal{R}) \le \sum_{i=k+1}^{K} R_i + R_k(-\mathcal{R}). \tag{4.6}$$

The first inequality holds because the red items marked $\mathcal{R}$ are in mixed bins by packing property 2. (If $\mathfrak{r}$ is not medium, $R_k(-\mathcal{R}) = R_k$, so it also holds.) By combining ineq. (4.5) and ineq. (4.6), we have

$$\textsc{UnmixedRed} \le \min\left\{ \sum_{i=k+1}^{K} R_i + R_k(-\mathcal{R}), \sum_{i=1}^{K} R_i - \sum_{i=k}^{K} B_i \right\}.$$

So if $\textsc{UnmixedRed} > 0$, the total number of bins in P' is at most

$$\begin{aligned} & \frac{1}{1-\varepsilon}\textsc{Tiny} + \textsc{UnmixedRed} + \sum_{i=0}^{K} B_i + O(1) \\ \le\ & B_0 + \frac{1}{1-\varepsilon}\textsc{Tiny} \\ & \quad + \min\left\{ \sum_{i=k+1}^{K} R_i + R_k(-\mathcal{R}) + \sum_{i=1}^{K} B_i, \sum_{i=1}^{K} R_i + \sum_{i=1}^{k-1} B_i \right\} + O(1). \end{aligned} \tag{4.7}$$

Let J be the set of types whose blue items are packed in pure blue bins, including type 1 and type N. For each item $\mathfrak{p}$ of type $t \neq N$, $t \in J$, we have $\text{leaves}(t) = 0 < k$, so $v_{k,s(\mathfrak{r})}(\mathfrak{p}) = \frac{1-\text{red}_t}{\text{bluefit}_t}$. Furthermore, $w_k(\mathfrak{p}) \ge \frac{1-\text{red}_t}{\text{bluefit}_t}$. We conclude $\sum_{j\in J}\sum_{t(\mathfrak{p}_i)=j} w_k(\mathfrak{p}_i) \ge \sum_{j\in J}\sum_{t(\mathfrak{p}_i)=j} v_{k,s(\mathfrak{r})}(\mathfrak{p}_i) \ge B_0 + \frac{\textsc{Tiny}}{1-\varepsilon}$ using packing property 5.

In the first term of the minimum in ineq. (4.7), we count all bins with blue items except the pure blue bins, all bins with red items of classes above k, and the bins with red items of class k that are not marked $\mathcal{R}$. (If red items of class k are small, this means all red items of this class.) This term is therefore upper bounded by $\sum_{j\notin J}\sum_{t(\mathfrak{p}_i)=j} w_k(\mathfrak{p}_i)$ (again using packing property 5). In the second term of the minimum in ineq. (4.7), we count all bins with red items, as well as bins with blue items of class at least 1 and at most $k-1$. The second term is therefore upper bounded by $\sum_{j\notin J}\sum_{t(\mathfrak{p}_i)=j} v_{k,s(\mathfrak{r})}(\mathfrak{p}_i)$. As noted above theorem 2, this is at most $\sum_{j\notin J}\sum_{t(\mathfrak{p}_i)=j} v_k(\mathfrak{p}_i)$. □

4.2.7 Offline solution

Having derived an upper bound for the total cost of an Extreme Harmonic algorithm in theorem 2, in order to calculate the asymptotic competitive ratio (definition 7), we now need to lower bound the optimal cost of a given input after post-processing. This will again depend on what the value of k is. There are two main cases if $k \in \{1,\dots,K\}$: $\mathfrak{r}$ is medium and $\mathfrak{r}$ is small. The case $k = K+1$ is much easier, because $w_{K+1}(\mathfrak{p}) \le v_{K+1}(\mathfrak{p})$ for each item $\mathfrak{p}$, so $\sum_{i=1}^{n} w_{K+1}(\mathfrak{p})$ upper bounds the cost of $\mathcal{A}$ by theorem 2, and this sum does not depend on any marks of items. We can therefore use a standard knapsack search as in Seiden [98] for this case and other papers.

For $k \in \{1, \dots, K\}$, we will be interested in the weights of items for a fixed value of k. It can be seen that in the range $(1/2, 1]$, the function $v_k(\mathfrak{p})$ changes at most once (viewed as a function of the size of $\mathfrak{p}$), namely at the threshold $1 - t_{t(\mathfrak{r})}$, where leaves(k) drops below k if $\mathfrak{r}$ is medium. On the other hand $w_k(\mathfrak{p}) = 1$ in the entire range $(1/2, 1]$. For a fixed value of $k < K + 1$, we therefore reduce the number of types again as follows. Recall that $t_3 = 1/2$, and $\mathfrak{r}$ is determined by k.

Case 1: $\mathfrak{r}$ is medium We set $t_2 = 1 - t_{t(\mathfrak{r})}$, $t_1 = 2/3$ and $t_0 = 1$. We set leaves$(2) = k$, leaves$(1) < k$ such that $\text{redspace}_{\text{leaves}(1)} = 1/3 < t_{t(\mathfrak{r})}$, and leaves$(0) = 0$.

Case 2: $\mathfrak{r}$ is small We set $t_2 = 2/3$ and $t_1 = 1$ as in Extreme Harmonic itself (requirement R(5)). We have $\text{redspace}_{\text{leaves}(2)} = 1/3$, and leaves$(1) = 0$.

After these changes, theorem 2 remains valid for any fixed k, as w_k and v_k remain unchanged (given k). This holds even though if $\mathfrak{r}$ is medium, the types do not match the types used by Extreme Harmonic; the important property is that they match the behavior of Extreme Harmonic for any fixed value of $k < K + 1$.

We now define patterns for the two main cases. Intuitively, a pattern describes the contents of a bin in the optimal offline solution. If $\mathfrak{r}$ is medium, a *pattern of class k* is an integer tuple $q = (q_0, q_1, \dots, q_{N-1}, (q^{\mathcal{N}}_{t(\mathfrak{r})}, q^{\mathcal{B}}_{t(\mathfrak{r})}, q^{\mathcal{R}}_{t(\mathfrak{r})}))$ where $q_i \in \mathbb{N} \cup \{0\}, q^{\mathcal{M}}_{t(\mathfrak{r})} \in \mathbb{N} \cup \{0\}$ for $\mathcal{M} \in \{\mathcal{N}, \mathcal{B}, \mathcal{R}\}$, $q^{\mathcal{N}}_{t(\mathfrak{r})} + q^{\mathcal{B}}_{t(\mathfrak{r})} + q^{\mathcal{R}}_{t(\mathfrak{r})} = q_{t(\mathfrak{r})}$ and

$$\sum_{i=0}^{N-1} q_i t_{i+1} < 1. \tag{4.8}$$

The values q_i describe how many items of type i are present in the bin. The value $q^{\mathcal{M}}_{t(\mathfrak{r})}$ counts the number of items of type $t(\mathfrak{r})$ and mark $\mathcal{M}$. It can be seen that any feasible packing of a bin can be described by a pattern: the only quantity that is not fixed by a pattern is the total size of the items of type N, which we will call sand. However, by ineq. (4.8), there can be at most $1 - \sum_{i=0}^{N-1} q_i t_{i+1}$ of sand in a bin packed according to pattern q. Conversely, for each pattern, a set of items matching the pattern that fit into a bin can be found by choosing the size of each item close enough (from above) to the lower bound t_{i+1} for its type; then ineq. (4.8) guarantees the items will fit.

If $\mathfrak{r}$ is small, we define a pattern of class k as an integer tuple $q = (q_1, \dots, q_{N-1})$ where $q_i \in \mathbb{N} \cup \{0\}$ and ineq. (4.8) holds using $q_0 = 0$ (note that the values t_1 and t_2 depend on whether $\mathfrak{r}$ is medium or small, but the definition of $t(\mathfrak{r})$ is consistent across these two cases).

There are only finitely many patterns for each value of k. Denote this set by $\mathcal{Q}_k$ for $k = 1, \dots, K$. If $\mathfrak{r}$ is small or $k = K + 1$, $\mathcal{Q}_k$ is a fixed set, denoted by $\mathcal{Q}$.

For a given weight function w of class k, we define the weight of pattern q, denoted as $w(q)$, as the sum of the weights of the non-sand items in it plus $w(1 - \sum_{i=0}^{N-1} q_i t_{i+1}, N, \emptyset)$. As noted, $1 - \sum_{i=0}^{N-1} q_i t_{i+1}$ is an upper bound for the amount of sand in a bin packed according to pattern q; this value is not necessarily in the range $(0, t_N]$. If $\mathfrak{r}$ is medium, $q_0 = 0$. Pattern q specifies all the information we need to calculate $w(q)$, as w does not depend on the precise size of non-sand items, and for class k we know exactly how many items there are (if any) for each mark.

We can describe the offline solution for a given post-processed input σ' by a distribution χ over the patterns, where $\chi(q)$ indicates which fraction of the bins in the

optimal solution are packed using pattern q. Theorem 1 shows that $\text{OPT}(\sigma') \leq \text{OPT}(\sigma)$, where σ refers to the original input and σ' refers to the input after post-processing.

To show that Extreme Harmonic has competitive ratio at most c for an input sequence σ with a particular value $k < K+1$, by theorem 2 it is sufficient to show that

$$\begin{aligned}\frac{\min\left\{\sum_{i=1}^{n} w_k(\mathfrak{p}_i), \sum_{i=1}^{n} v_k(\mathfrak{p}_i)\right\}}{\text{OPT}(\sigma')} &= \min\left\{\frac{\sum_{i=1}^{n} w_k(\mathfrak{p}_i)}{\text{OPT}(\sigma')}, \frac{\sum_{i=1}^{n} v_k(\mathfrak{p}_i)}{\text{OPT}(\sigma')}\right\} \\ &\leq \min\left\{\sum_{q\in\mathcal{Q}_k} \chi(q) w_k(q), \sum_{q\in\mathcal{Q}_k} \chi(q) v_k(q)\right\} \leq c\end{aligned} \tag{4.9}$$

for all such inputs σ, using $\sum_{i=1}^{n} w(\mathfrak{p}_i) \leq \text{OPT}(\sigma') \sum_{q\in\mathcal{Q}_k} \chi(q) w(q)$ for $w \in \{w_k, v_k\}$, as $w(q)$ uses an upper bound for the amount of sand but is otherwise exactly the sum of the weights of the items in it.

As can be seen from this bound, the question now becomes: what is the distribution χ (the mix of patterns) that maximizes the minimum in ineq. (4.9)? We begin by deriving some crucial constraints on χ for the important case that $\mathfrak{r}$ is medium. This is the point where we start using the marks. The notation $q_i(q)$ refers to entry q_i in pattern q. We use $q^{-\mathcal{B}}_{t(\mathfrak{r})}(q)$ as shorthand for $q^{\mathcal{R}}_{t(\mathfrak{r})}(q) + q^{\mathcal{N}}_{t(\mathfrak{r})}(q)$.

We define three important patterns q^1, q^2, q^3. For $i = 1, 2, 3$, let

$$q^i = (0, 1, 0, \ldots, 0, 1, 0, \ldots, 0, (e_i)),$$

where the second 1 is at position $t(\mathfrak{r})$, and e_i is the i-th three-dimensional unit vector. These are the three possible patterns with an item of type $t(\mathfrak{r})$ and an item larger than $1 - s(\mathfrak{r})$. By requirement R(8), no non-sand item can be added to any of these patterns while maintaining $\sum_{i=0}^{N-1} q_i t_{i+1} < 1$.

Lemma 9. *If $\mathfrak{r}$ is medium, then*

$$\chi(q^1) \leq \frac{1 - \text{red}_{t(\mathfrak{r})}}{1 + \text{red}_{t(\mathfrak{r})}} \sum_{q \neq q^1} \chi(q) q^{-\mathcal{B}}_{t(\mathfrak{r})}(q).$$

Proof. We ignore additive constants in this proof, as we will divide by $\text{OPT}(\sigma')$ at the end to achieve our result. The pattern q^1 contains an $\mathcal{N}$-item that is strictly smaller than $\mathfrak{r}$. We apply lemma 5 for $i = t(\mathfrak{r})$ (ignoring the additive constant) to get

$$\begin{aligned}\chi(q^1)\text{OPT}(\sigma') &\leq \frac{1 - \text{red}_{t(\mathfrak{r})}}{2}(n^{t(\mathfrak{r})}(\mathcal{R}) + n^{t(\mathfrak{r})}(\mathcal{N})) \\ &\leq \frac{1 - \text{red}_{t(\mathfrak{r})}}{2}\left(\chi(q^1) + \sum_{q\neq q^1} \chi(q) q^{-\mathcal{B}}_{t(\mathfrak{r})}(q)\right)\text{OPT}(\sigma'),\end{aligned}$$

and the bound in the lemma follows. □

Lemma 10. *In q^2, the $\mathcal{B}$-item $\mathfrak{p}$ of type $t(\mathfrak{r})$ is blue.*

Proof. Extreme Harmonic did not pack $\mathfrak{p}$ alone in a bin as a red item, since it is smaller than $\mathfrak{r}$. But by packing property 7, $\mathfrak{p}$ also was not packed in a mixed bin as a red $\mathcal{B}$-item. □

We will use the following notation.

Definition 15. *Let* $\text{RedComp}(i) = \{j < N \mid 0 < \text{needs}(j) \leq \text{leaves}(i)\}$ *be the set of all types* j *such that red items of type* j *can be packed with blue items of type* i*.*

Lemma 11. *If* $\mathfrak{r}$ *is medium, then*

$$\frac{1}{2}\chi(q^2) \leq \sum_{j \in \text{RedComp}(t(\mathfrak{r}))} \sum_{q} \rho_j \chi(q) q_j(q) \,.$$

Proof. As before, we ignore additive constants. There are $\chi(q^2)\text{OPT}(\sigma')$ bins packed with pattern q^2, meaning that σ' contains at least $\chi(q^2)\text{OPT}(\sigma')$ blue $\mathcal{B}$-items of type $t(\mathfrak{r})$ by lemma 10. So in the packing P', there exist at least $\frac{1}{2}\chi(q^2)\text{OPT}(\sigma')$ bins with two blue $\mathcal{B}$-items of type $t(\mathfrak{r})$ and by packing property 2 these bins are mixed and contain red items. The red items are red-compatible with those $\mathcal{B}$-items. That is, each such red item is of a type $j \in \text{RedComp}(t(\mathfrak{r}))$.

The number of items of type j in σ' is given by $\sum_q \chi(q) q_j(q) \cdot \text{OPT}(\sigma')$. By theorem 1, the number of red items of type j is $\text{red}_j \sum_q \chi(q) q_j(q) \cdot \text{OPT}(\sigma')$. We place redfit_j red items together in each bin by packing property 5. This means that the number of bins in P with red items of type j is $\frac{\text{red}_j}{\text{redfit}_j} \sum_q \chi(q) q_j(q) \cdot \text{OPT}(\sigma') = \rho_j \sum_q \chi(q) q_j(q) \cdot \text{OPT}(\sigma')$. Summing over all types $j \in \text{RedComp}(t(\mathfrak{r}))$, we find that

$$\begin{aligned}
\frac{1}{2}\chi(q^2)\text{OPT}(\sigma') \leq \quad & (\text{number of bins in } P' \text{ with} \\
& \text{two blue } \mathcal{B}\text{-items of type } t(\mathfrak{r}) \text{ and red items}) \\
\leq \quad & (\text{number of bins in } P' \text{ with} \\
& \text{red items that fit with items of type } t(\mathfrak{r})) \\
= \quad & \left(\sum_{j \in \text{RedComp}(t(\mathfrak{r}))} \sum_{q} \rho_j \chi(q) q_j(q) \right) \cdot \text{OPT}(\sigma').
\end{aligned}$$

□

4.2.8 Linear program

Maximizing the minimum in ineq. (4.9) is the same as maximizing the first term under the condition that it is not larger than the second term – except that this condition might not be satisfiable, in which case we need to maximize the second term. For each value of $k \in \{1, \ldots, K\}$, we will therefore consider two linear programs, and furthermore these linear programs will differ depending on whether $\mathfrak{r}$ is medium or small, so that in total we get four different LPs which we will call $P_w^{k,\text{med}}$,$P_w^{k,\text{sml}}$, $P_v^{k,\text{med}}$ and $P_v^{k,\text{sml}}$ (we will use the notation P_w^k (P_v^k) whenever we want to refer to both $P_w^{k,\text{med}}$ and $P_w^{k,\text{sml}}$ ($P_v^{k,\text{med}}$ and $P_v^{k,\text{sml}}$)). Let $\mathcal{Q}_k = \{q^1, \ldots, q^{|\mathcal{Q}_k|}\}$ and let $\chi_i = \chi(q^i), w_{ik} = w_k(q^i), v_{ik} = v_k(q^i), n_{ij} = q_j(q^i), m_i = q_{t(\mathfrak{r})}^{-\mathcal{B}}(q^i)$. If $\mathfrak{r}$ is medium, $P_w^{k,\text{med}}$ is the following linear program.

$$\left(P_w^{k,\text{med}}\right) \qquad \begin{aligned} \max \quad & \sum_{i=1}^{|\mathcal{Q}_k|} \chi_i w_{ik} \\ \text{s.t.} \quad & \chi_1 - \frac{1-\text{red}_{t(\mathfrak{r})}}{1+\text{red}_{t(\mathfrak{r})}} \sum_{i=3}^{|\mathcal{Q}_k|} \chi_i m_i \le 0 && (4.10) \\ & \frac{1}{2}\chi_2 - \sum_{j \in \text{RedComp}(t(\mathfrak{r}))} \sum_{i=3}^{|\mathcal{Q}_k|} \rho_j \chi_i n_{ij} \le 0 && (4.11) \\ & \sum_{i=3}^{|\mathcal{Q}_k|} \chi_i (w_{ik} - v_{ik}) \le 0 && (4.12) \\ & \sum_{i=1}^{|\mathcal{Q}_k|} \chi_i \le 1 && (4.13) \\ & \chi(q) \ge 0 \qquad \forall q \in \mathcal{Q}_k && (4.14) \end{aligned}$$

$P_w^{k,\text{med}}$ has a very large number of variables but only four constraints (apart from the nonnegativity constraints). Constraint (4.10) is based on lemma 9, where we have used that q^2 does not contain any item marked $\mathcal{N}$ or $\mathcal{R}$, implying $m_2 = 0$. Constraint (4.11) is based on lemma 11, using that q^1 and q^2 do not contain non-sand items of size less than 1/3, so $n_{1j} = 0$ and $n_{2j} = 0$ for all j for which $\text{needs}(j) \le \text{leaves}(t(\mathfrak{r}))$.[9] Constraint (4.12) simply says that the objective function must be at most $\sum_{i=1}^{|\mathcal{Q}_k|} \chi_i v_{ik}$ (using that $w_{ik} = v_{ik}$ for $i = 1, 2$, which we will prove in lemma 12): if this does not hold, we should be solving the linear program $P_v^{k,\text{med}}$, which has objective function $\sum_{i=1}^{|\mathcal{Q}_k|} \chi_i v_{ik}$, instead. The final constraints (4.13) and (4.14) say that χ is a distribution.

Lemma 12. $v_{1k} = w_{1k} = w_{2k} = v_{2k}$.

Proof. Recall that q^1 contains one $\mathcal{N}$-item of type $t(\mathfrak{r})$, i.e. the same type as $\mathfrak{r}$, and one item larger than $1 - s(\mathfrak{r})$. Call the $\mathcal{N}$-item $\mathfrak{r}'$ and the large one $\mathfrak{L}$; note that $t(\mathfrak{L}) = 2$. We have that $w_k(q^1) = w_k(\mathfrak{r}') + w_k(\mathfrak{L}) + S$, where S is an upper bound for the weight of the sand, and $v_k(q^1) = v_k(\mathfrak{r}') + v_k(\mathfrak{L}) + S$ (the maximum possible amount of sand and hence also its weight is equal in the two cases). As $\text{red}_2 = 0$ ($\mathfrak{L}$ is larger than 1/2 and such items are never red), and $\mathfrak{L}$ is too large to be combined with $\mathfrak{r}$, we have $w_k(\mathfrak{L}) = v_k(\mathfrak{L}) = 1/\text{bluefit}_2 = 1$.

For $w_k(\mathfrak{r}')$, consider that $\mathfrak{r}'$ and $\mathfrak{r}$ have the same type, and as the mark of $\mathfrak{r}'$ is $\mathcal{N}$, we get $w_k(\mathfrak{r}') = \beta_{t(\mathfrak{r})} + \rho_{t(\mathfrak{r})} = \frac{1-\text{red}_{t(\mathfrak{r})}}{\text{bluefit}_{t(\mathfrak{r})}} + \frac{\text{red}_{t(\mathfrak{r})}}{\text{redfit}_{t(\mathfrak{r})}}$. For type $t(\mathfrak{r})$, we have that $\text{leaves}(t(\mathfrak{r})) < \text{needs}(t(\mathfrak{r}))$ (property 2). Therefore, $v_k(\mathfrak{r}') = \beta_{t(\mathfrak{r})} + \rho_{t(\mathfrak{r})} = w_k(\mathfrak{r}')$. This shows that $v_{1k} = w_{1k}$.

The pattern q^2 contains one $\mathcal{B}$-item of type $t(\mathfrak{r})$ (denoted by $\mathfrak{r}''$) and one item larger than $1 - \mathfrak{r}$ (again denoted by $\mathfrak{L}$). We have $w_k(\mathfrak{r}'') = w_k(\mathfrak{r}')$ since the weight w_k is the same for $\mathcal{N}$- and $\mathcal{B}$-items of the same type. As above, we find $w_k(\mathfrak{L}) = v_k(\mathfrak{L}) = 1$

[9] We also have $n_{3j} = 0$, but we keep the term for $i = 3$ in constraint (4.11) to make the dual easier to write down.

and $v_k(\mathfrak{r}'') = \beta_{t(\mathfrak{r})} + \rho_{t(\mathfrak{r})} = \frac{1-\mathrm{red}_{t(\mathfrak{r})}}{\mathrm{bluefit}_{t(\mathfrak{r})}} + \frac{\mathrm{red}_{t(\mathfrak{r})}}{\mathrm{redfit}_{t(\mathfrak{r})}} = w_k(\mathfrak{r}'')$. This shows $w_{2k} = v_{2k}$ and $w_{1k} = w_{2k}$. □

For the case when $\mathfrak{r}$ is small, we do not have constraints (4.10) and (4.11), and the linear program $P_w^{k,\mathrm{sml}}$ looks as follows. Here we denote the set of patterns simply by $\mathcal{Q}$ since it is the same for all values of k for which $\mathrm{redspace}_k \le 1/3$. In this setting, q^1, q^2, q^3 do not have a special meaning.

$$\left(P_w^{k,\mathrm{sml}}\right) \quad \begin{aligned} \max \quad & \textstyle\sum_{i=1}^{|\mathcal{Q}|} \chi_i w_{ik} && && \\ \text{s.t.} \quad & \textstyle\sum_{i=1}^{|\mathcal{Q}|} \chi_i (w_{ik} - v_{ik}) \le 0 && && (4.15)\\ & \textstyle\sum_{i=1}^{|\mathcal{Q}|} \chi_i \le 1 && && (4.16)\\ & \chi_i \ge 0 && \forall i = 1,\dots,|\mathcal{Q}| && (4.17) \end{aligned}$$

Intermezzo It is useful to consider the value of w_{1k} (etc.). We have not discussed the values of the parameters yet. However, as an example, for the algorithm Harmonic++, two of the types are $(341/512, 1]$ and $(1/3, 171/512]$ (types 1 and 18). Let us consider the case where at the end of the input, an item of type 18 is alone in a bin, and no smaller items are alone in bins. For this case, for Harmonic++, the two weighting functions for the pattern which contains types 1 and 18 both evaluate to

$$1 + \frac{1 - 0.176247}{2} + \frac{0.176247}{1} + \frac{1}{1 - \frac{1}{50}} \cdot \frac{1}{1536} \approx 1.58879.$$

In other words, a distribution χ consisting only of this one pattern immediately gives a lower bound of 1.58879 on the competitive ratio of Harmonic++.

Our improved packing of the medium items and our marking of them ensures that this distribution, where the optimal solution uses critical bins exclusively, can no longer be used, since it is not a feasible solution to $P_w^{k,\mathrm{med}}$. This is the key to our improvement over Harmonic++.

Dual program

Our general idea is as follows: We consider the duals of the linear programs given above. These dual LPs have variables $y_1, \dots, y_4$ or y_3, y_4, respectively. Any feasible solution for the dual (which is a minimization problem) is an upper bound on the competitive ratio of our algorithm by duality and by ineq. (4.9). We are interested in feasible dual solutions with objective value c, where c is our target competitive ratio. Our goal is then to find feasible values for y_1, y_2 and y_3 (or only y_3) such that the dual becomes feasible.

Case 1: $\mathfrak{r}$ is small The dual of $P_w^{k,\mathrm{sml}}$ is the following.

$$\left(D_w^{k,\mathrm{sml}}\right) \quad \begin{aligned} \min \quad & y_4 && && \\ \text{s.t.} \quad & (w_{ik} - v_{ik}) y_3 + y_4 \ge w_{ik} && i = 1,\dots,|\mathcal{Q}| && (4.18)\\ & y_i \ge 0 && i = 3, 4 && \end{aligned}$$

If constraint (4.18) does not hold for pattern q_i and a given dual solution y^*, we have

$$(1 - y_3^*)w_{ik} + y_3^* v_{ik} > y_4^* \tag{4.19}$$

We need to determine if there is a pattern such that ineq. (4.19) holds. For $y_3^* \in [0,1]$, the left hand side of ineq. (4.19) represents a weighted average of the weights w_{ik} and v_{ik}. We add the condition $y_3 \leq 1$ to $D_w^{k,\text{sml}}$. A feasible solution with objective value c and $y_3 \leq 1$ exists for $D_w^{k,\text{sml}}$ if and only if a feasible solution with objective value c and $y_3 \leq 1$ exists for $D_v^{k,\text{sml}}$, as ineq. (4.19) is now symmetric in w and v. This means that feasibility of $D_w^{k,\text{sml}}$ and $D_v^{k,\text{med}}$ with $y_3 \leq 1$ can be checked at the same time. Again, note that it is sufficient for our purposes to find a feasible solution.

Definition 16. *In case $D_w^{k,sml}$ is used, we define $\omega_k(\mathfrak{p}) = (1 - y_3^*)w_k(\mathfrak{p}) + y_3^* v_k(\mathfrak{p})$.*

We define $\omega_k(\mathfrak{p})$ as given in definition 16 for each item $\mathfrak{p}$. Since $\mathfrak{r}$ is small, there are no marked items of type $t(\mathfrak{r})$, so $\omega_k(\mathfrak{p})$ depends only on the type and size of $\mathfrak{p}$. The problem of determining $W = \max_{q \in \mathcal{Q}} \omega_k(q)$ for a given value of y_3^* is a simple knapsack problem, which is straightforward to solve using dynamic programming.

All that remains to be done is to determine a value for y_3^* for given k such that $W \leq c$. The values that were used for Son Of Harmonic can be found in table 4.2 on page 61. In order to find these values, we used a binary search in the interval $[0,1]$. We start by setting $y_3^* = 1/2^{10}$ and compute W. If $W \leq y_4^*$, $D_w^{k,\text{sml}}$ and $D_v^{k,\text{sml}}$ have objective value at most y_4^* and we are done. Else, the dynamic program returns a pattern q such that $\omega_k(q) > y_4^*$. For this pattern q, we compare its weights according to w and v. If $w_{ik} > v_{ik}$, we increase y_3^*, else we decrease it (halving the size of the interval we are considering). If after 20 iterations we still have no feasible solution, we return infeasible. This may be incorrect (it depends on how long we search), but our claimed competitive ratio depends only on the correctness of *feasible* solutions.

Let us summarize the above discussion. If $\mathfrak{r}$ is small, proving that an Extreme Harmonic algorithm is c-competitive can be done by running the binary search for $k = \text{needs}(t^*)$ using $y_4^* = c$. If successful, this yields some value y_3^*. If (y_3^*, y_4^*) is a feasible solution for $D_w^{k,\text{sml}}$, then $(1 - y_3^*, y_4^*)$ is a feasible solution for $D_v^{k,\text{sml}}$.

Case 2: $\mathfrak{r}$ is medium For the more interesting case when $\mathfrak{r}$ is medium, the dual $D_w^{k,\text{med}}$ of the program $P_w^{k,\text{med}}$ is the following.

$$\begin{aligned}
\min \quad & y_4 \\
\text{s.t.} \quad & y_1 + y_4 \geq w_{1k} && (4.20)\\
& \frac{1}{2} y_2 + y_4 \geq w_{2k} && (4.21)\\
\left(D_w^{k,\text{med}}\right) \quad & -\frac{1 - \text{red}_{t(\mathfrak{r})}}{1 + \text{red}_{t(\mathfrak{r})}} m_i y_1 - y_2 \sum_{j \in \text{RedComp}(t(\mathfrak{r}))} \rho_j n_{ij} \\
& + (w_{ik} - v_{ik}) y_3 + y_4 \geq w_{ik} \qquad i = 3, \ldots, |\mathcal{Q}_k| && (4.22)\\
& y_i \geq 0 \qquad i = 1,2,3,4 && (4.23)
\end{aligned}$$

[10] In fact, we did start at $y_3^* = 3/16$ for Son Of Harmonic. This was only to speed up computations as soon as we noticed that such low values work well.

Again we restrict ourselves to solutions with $y_3^* \in [0,1]$. If the value $y_4^* = c \geq w_{1k} = w_{2k}$, then constraints (4.20) and (4.21) are automatically satisfied by constraint (4.23). In this case we can set $y_1^* = 0$ and $y_2^* = 0$. In effect, this reduces $D_w^{k,\text{med}}$ to $D_w^{k,\text{sml}}$, for which we already know how to find a feasible value for y_3^*. We therefore ignore the entire marking done by the algorithm and set the weight for each item to be the weight for the case that its mark is not $\mathcal{R}$. Then weights again do not depend on marks and we apply the method from Case 1.

Let us now consider the case $y_4^* = c < w_{1k}$.[11] For given y_4^* we need to determine if $D_w^{k,\text{med}}$ and $D_v^{k,\text{med}}$ are feasible; this requires finding suitable values for y_1, y_2 and y_3. If a solution vector y^* is feasible for $D_w^{k,\text{med}}$ (or $D_v^{k,\text{med}}$), $y_4^* < w_{1k} = v_{1k} = w_{2k} = v_{2k}$, and constraint (4.20) or constraint (4.21) is not tight, then we can decrease y_1^* and/or y_2^* and still have a feasible solution. We therefore restrict our search to solutions for which constraints (4.20) and (4.21) are tight, and $y_4^* < w_{1k}$. Then

$$y_1^* = w_{1k} - y_4^* \qquad > 0 \tag{4.24}$$

$$y_2^* = 2(w_{1k} - y_4^*) \qquad > 0. \tag{4.25}$$

This means that given $y_4^* < w_{1k}$, we know the values of y_1^* and y_2^*. We can therefore prove y_4^* is a feasible objective value for $D_w^{k,\text{med}}$ by giving y_3^*-values that make the linear program feasible. If constraint (4.22) does not hold for pattern q_i ($i \geq 3$) and a given dual solution y^*, we have the following by some simple rewriting:

$$(1 - y_3^*)w_{ik} + y_3^* v_{ik} + \frac{1 - \text{red}_{t(\mathfrak{r})}}{1 + \text{red}_{t(\mathfrak{r})}} m_i y_1^* + y_2^* \sum_{j \in \text{RedComp}(t(\mathfrak{r}))} \rho_j n_{ij} > y_4^* \tag{4.26}$$

If this holds for some pattern q that contains an $\mathcal{R}$-item, then it obviously also holds if we replace that $\mathcal{R}$-item by an $\mathcal{N}$-item of the same type. This gives a pattern with the same values $m_i = q_{-\mathcal{B}}^{t(\mathfrak{r})}(q^i)$ and $n_{ij} = q_j(q^i)$ but a higher value for w_{ik}. It is therefore sufficient to check the patterns with $\mathcal{N}$-items. The only exception to this is if replacing the $\mathcal{R}$-item by an $\mathcal{N}$-item would give pattern q^1, which does have weight larger than y_4^* and therefore violates ineq. (4.26) (but constraint (4.22) does not involve pattern q^1). We therefore check pattern q^3 separately in the program.

We can now define a new weighting function $\omega_k(\mathfrak{p})$, which depends only on types and sizes (and not on marks).

Definition 17. *In case $D_w^{k,med}$ is used, we define*

$$\omega_k(\mathfrak{p}) = \begin{cases} (1 - y_3^*)\left(\beta_{t(\mathfrak{r})} + \rho_{t(\mathfrak{r})}\right) + y_3^* v_k(\mathfrak{p}) + \frac{1-\text{red}_{t(\mathfrak{r})}}{1+\text{red}_{t(\mathfrak{r})}} y_1^* & \text{if } t(\mathfrak{p}) = t(\mathfrak{r}) \\ (1 - y_3^*)w_k(\mathfrak{p}) + y_3^* v_k(\mathfrak{p}) + \rho_{t(\mathfrak{p})} y_2^* & \text{if } t(\mathfrak{p}) \in \text{RedComp}(t(\mathfrak{r})) \\ \frac{1}{1-\varepsilon} x & \text{if } t(\mathfrak{p}) = N \\ (1 - y_3^*)w_k(\mathfrak{p}) + y_3^* v_k(\mathfrak{p}) & \text{else} \end{cases}$$

In order to prove that an Extreme Harmonic algorithm is c-competitive if $\mathfrak{r}$ is medium and $c < w_{1k}$, it is sufficient to verify that there exists a value $y_3^* \in [0,1]$

[11] This means that the critical patterns q^1, q^2 have weight higher than our target competitive ratio. These are exactly the cases, where Seiden's simpler analysis reaches its limits (see the Intermezzo) and where we can use our markings for better results.

such that $\max_{q \in Q_k} \omega_k(q) \leq c$. Values for y_3^* that satisfy this can be found in table 4.2. Finding these values was done by a binary search for each value of k for which $\text{redspace}_k > 1/3$, each time setting $y_4^* = c$ and using inequalities (4.24) and (4.25).

Summary Overall, our approach is as follows: We first fix a target competitive ratio c. We do the following for every value of $k \in \{1, \ldots, K\}$. Consider the value for y_3^* (for our algorithm Son Of Harmonic, these values are specified in table 4.2). If $\mathfrak{r}$ is small, we check that $D_w^{k,\text{sml}}$ is feasible for $y_4^* = c$ and this y_3^*. If $\mathfrak{r}$ is medium, we compute w_{1k} and check whether $w_{1k} \leq c$ or $w_{1k} > c$. In the latter case, we again check that $D_w^{k,\text{sml}}$ is feasible for $y_4^* = c$ and the given value of y_3^*. In the former case, we check that $D_w^{k,\text{med}}$ is feasible for $y_4^* = c$, the given value of y_3^*, and y_1^*, y_2^* as given by inequalities (4.24) and (4.25). Finally, for $k = K + 1$, it is sufficient to count blue bins, and we solve a single knapsack problem based on w_k alone, checking that the heaviest pattern is not heavier than $y_4^* = c$. A pseudocode description of this can be found in algorithm 8.

Algorithm 8 The procedure for checking a certain target competitive ratio for an Extreme Harmonic algorithm. The procedure used for solving the knapsack problem is described in algorithm 9.

```
Read parameters of the Extreme Harmonic algorithm
blueWeight[i] ← (1−red_i)/bluefit_i
maxWeight ← solveKnapsack(blueWeight)                    // Check case k = K + 1
if maxWeight > targetRatio then return false
end if
for k = 0,...,K do
    // Find the correct weighting function to be used
    if r is medium then
        Compute w_{1k}
        if w_{1k} < targetRatio then
            ω_k ← weight function according to definition 16
        else
            ω_k ← weight function according to definition 17
        end if
    else
        ω_k ← weight function according to definition 16
    end if
    // Now solve the knapsack problem and compare with target ratio
    maxWeight ← solveKnapsack(ω_k)
    if maxWeight > targetRatio then return false
    end if
end for
// In all cases, maxWeight ≤ targetRatio, so we verified this competitive ratio
return true
```

Solving the knapsack problems

In order to prove our competitive ratio $c = 1.5813$, we prove feasibility of the discussed dual linear programs, which amounts to solving knapsack problems and comparing the maximum weight of a pattern to our target competitive ratio. We will now describe how our implementation of this knapsack solving works, given a set of item types as described at the beginning of section 4.2.7 and a corresponding weight function w (one weight per type). See also algorithm 9.

We use two main heuristics to speed up the computation. First, for each type i, we define the expansion $\exp_i$ of type i as the weight according to function w divided by t_{i+1}. Now we sort the types in decreasing order of expansion; call this permutation of types π. When constructing a pattern with high weight, we try to add items in the order of this permutation. Note that π will not contain types that have expansion below that of sand: Such types will not be part of a maximum weight pattern, as the pattern with sand instead of these items has no smaller weight.

Second, we use branch and bound. We use a variable maxFound that will store the maximum weight of a pattern found so far, and give this the initial value $c - 1/1000$. Whenever the current pattern cannot be extended to a pattern with weight more than maxFound (based on the expansion of the next item in the ordering π that still fits), we stop the calculation for this branch. Initializing maxFound with a value close to c immediately eliminates many patterns.

The process works as follows. We start with type $t = \pi(1)$ (i.e., the type with the largest expansion) and an empty pattern. For current type $t = \pi(j)$ and current pattern q that contains non-sand items of total size S and total weight $w(q)$, we compute an upper bound on the weight that this pattern q can at most get by adding items of types $\pi(j), \pi(j+1), \ldots$, as follows. We find the first type i in this order that still fits with the items of q and compute $u = w(q) + (1 - S)\exp_i$. This is an upper bound for the weight of any bin which contains the items from q, as it assumes we could fill up the whole remaining space $1 - S$ in the bin with items of expansion $\exp_i$. If this upper bound is already smaller than maxFound, we immediately cancel the further exploration of this pattern q.

Otherwise, if we have no more types to add (i.e. we reached the end of list of types in π), set maxFound to the weight of q (including the sand) and store q as the heaviest pattern so far. If we still have more types to explore, find out how many items of the next type can fit maximally into q; call this number m (if adding an item of the next type would create pattern q^1 or q^2 and we are considering the dual program $D_w^{k,\text{med}}$, we set $m = 0$ as we do not need to consider these patterns). Now recursively call this procedure with type $\pi(j+1)$ and patterns $q_0, \ldots, q_m$ where q_i is obtained from q by adding i items of type t.

The heuristics described in this section are still not enough to be able to examine all possible patterns in reasonable time if the number of types is too large. We explain in the next section how to reduce the set of patterns further by reducing the number of small types and how to ensure that larger items are more important than smaller items (by making sure the expansion of small items is monotonically nondecreasing in the size, that is, larger (but still small) items do not have smaller expansions than smaller items).

Algorithm 9 solveKnapsack(w) Solving a knapsack problem for a given weight function w. The procedure packRecursively(i, p) is described in algorithm 10.

```
1: p ← new pattern object
2: maxFound ← targetRatio −0.001
3: heaviestPattern ← null
4: Create permutation according to expansions
5: packRecursively(0, p)
6: return heaviestPattern
```

Algorithm 10 packRecursively(i, p) We denote by $w(p)$ the total weight of non-sand items in p and by $S(p)$ the free space in p. N' denotes the number of types in the permutation.

```
1: exp ← maximum expansion of items fitting in p
2: ub ← w(p) + exp · S(p)
3: if maxFound > ub then return          // No need to explore this pattern further
4: else if i = N' then                   // We tested all types, so we are done
5:     maxFound ← ub
6:     heaviestPattern ← p
7: else
8:     t ← i-th type in permutation      // this is the next type we want to add
9:     m ← S(p)/t_{t+1}                  // how many type t items can we add?
10:    if adding type t to p creates pattern q_1 then
11:        m ← 0                         // Do not add this type
12:    end if
13:    while m ≥ 0 do
14:        p' ← pattern obtained by adding m type t items to p
15:        // Continue with next type in permuted order
16:        packRecursively(i + 1, p')
17:        m ← m − 1
18:    end while
19: end if
```

4.3 The algorithm Son Of Harmonic

For our algorithm Son Of Harmonic we have set initial values as follows. The right part of table 4.1 below contains item sizes and corresponding red_i values that were set manually. Some numbers of the form $1/i$ until the value t_N are added automatically by our program if they are not listed below (see below for details on how these are selected).

The remaining values red_i are set automatically using heuristics designed to speed up the search and minimize the resulting upper bound. In the range $(1/3, 1/2]$, we automatically generate item sizes (with corresponding values red_i and redspace_i) that are less than t_N apart to ensure uniqueness of q^1 and q^2: no non-sand item can be packed into any bin of pattern q^1 or q^2. The value Γ specifies an upper bound on how much room is used by red items of size at most $1/14$; larger items ($\leq 1/3$) use at most $1/3$ room. Since we have this bound Γ, we also add type thresholds of the form Γ/i for $i = 1, 2, 3, 4$, to ensure that items just below this threshold can be packed without leaving much space unused.

Table 4.1: Parameters and item types used for Son Of Harmonic.

(a) Parameters

Parameter	Value
c	$\frac{15813}{10000}$
t_N	$\frac{1}{4000}$
Γ	$\frac{2}{7}$ (starting from $\frac{1}{14}$)
$\mathcal{T}$	$\frac{1}{50}$

(b) Size lower bounds and values red_i

Item size	red_i
33345/100000	0
33340/100000	0
33336/100000	0
33334/100000	0
5/18	2/100
7/27	105/1000
1/4	1061/10000
8/39	8/100
1/5	93/1000
3/17	3/100
1/6	8/100
3/20	0
29/200	0
1/7	16/100

The last parameter is some item size $\mathcal{T} = t_j$. Above this size, we generate all item sizes of the form $1/i$ for $i > 3$. Below this size, we skip some item sizes as described below.

Our program uses an exact representation of fractions, with numerators and denominators of potentially unbounded size, in order to avoid rounding errors. The source code and the full list of all types and parameters as determined by the program can be found at `https://sheydrich.github.io/ExtremeHarmonic/`. In appendix A, we provide an alternative set of parameters, which give a competitive ratio of 1.583 with a much smaller set of knapsack problems to check.

Additionally, in table 4.2 we provide the y_3^*-values that certify the competitive ratio of our algorithm.

Automatic generation of item sizes We start by generating all item sizes of the form $1/i$ for i between 2 and $\mathcal{T}$ (if they are not already present in the parameter file). After that, we generate types above 1/3 in steps of size t_N. By choosing this step size, we make sure that no non-sand items can be added to the patterns q^1, q^2, q^3. The value red_j for such a type j is chosen such that the pattern containing an item $\mathfrak{r}'$ of type j and a large item $\mathfrak{L}$ of type 2 (i.e., $t_{i+1} = 1/2$) has as weight exactly our target competitive ratio if $k = K + 1$. That is, we consider the weighting function w_{K+1}. We have $w_{K+1}(\mathfrak{r}') = \frac{1-\mathrm{red}_j}{2}$, $w_{K+1}(\mathfrak{L}) = 1$, and an upper bound for the amount of sand that fits with these items is $1/2 - t_{j+1}$. Therefore, red_j is defined as the solution of the equation

$$1 + \frac{1-\mathrm{red}_j}{2} + \frac{1}{1-\varepsilon}\left(\frac{1}{2} - t_{j+1}\right) = c = 1.5813, \tag{4.27}$$

as long as this value is positive. We stop generating types as soon as it becomes negative. To be precise, our highest value t_{j+1} is defined by taking $\mathrm{red}_j = 0$ in eq. (4.27).

Table 4.2: y_1^*-, y_2^*- and y_3^*-values used to certify that Son Of Harmonic is 1.5813-competitive. In cases where no y_1^*-value is given, $D_w^{k,\text{sml}}$ was used. Note that only two different values for y_3^* were used.

k	$y_1^* = \frac{y_2^*}{2}$	y_3^*
≤ 4	–	$\frac{9}{32}$
5	–	$\frac{3}{16}$
6, 7	–	$\frac{9}{32}$
8, …, 43	–	$\frac{3}{16}$
44, …, 49	–	$\frac{9}{32}$
51	$\frac{246839}{59985000}$	$\frac{3}{16}$
52, 53	$\frac{27471}{6665000}$	$\frac{3}{16}$
54	$\frac{27271}{6665000}$	$\frac{3}{16}$
55	$\frac{80813}{19995000}$	$\frac{3}{16}$
56	$\frac{83813}{19995000}$	$\frac{3}{16}$
57	$\frac{26271}{6665000}$	$\frac{3}{16}$
58	$\frac{73813}{19995000}$	$\frac{3}{16}$
59	$\frac{68813}{19995000}$	$\frac{3}{16}$
60	$\frac{21271}{6665000}$	$\frac{3}{16}$
61	$\frac{58813}{19995000}$	$\frac{3}{16}$
62	$\frac{53813}{19995000}$	$\frac{3}{16}$
63	$\frac{16271}{6665000}$	$\frac{3}{16}$
64	$\frac{43813}{19995000}$	$\frac{3}{16}$
65	$\frac{38813}{19995000}$	$\frac{3}{16}$
66	$\frac{11271}{6665000}$	$\frac{3}{16}$
67	$\frac{28813}{19995000}$	$\frac{3}{16}$
68	$\frac{23813}{19995000}$	$\frac{3}{16}$
69	$\frac{6271}{6665000}$	$\frac{3}{16}$
70	$\frac{13813}{19995000}$	$\frac{3}{16}$
71	$\frac{8813}{19995000}$	$\frac{3}{16}$
72	$\frac{41}{215000}$	$\frac{3}{16}$
> 72	–	$\frac{3}{16}$

We have now generated all item sizes above $\mathcal{T}$. We generate large types as described in section 4.2.8. In the range $(\mathcal{T}, t_N)$, we do not generate all $1/i$ types, but we skip some (to speed up the knapsack search) if this can be done without a deterioriation in the competitive ratio. We do this by considering the expansion of such items, that is, the weight divided by the infimum size. We will ensure that the expansion of smaller items is smaller than that of larger items, so that they are irrelevant (or less relevant) for the knapsack problem.

Let us consider how we test whether a certain type $(1/j, x]$ is required (where x is the next larger type, i.e. either the last type generated before we started this last phase or the last type generated in this phase), and which red_i we should choose. Denote by $s_i := 1/j$ the value we want to check. We compute a lower and upper bound $\underline{\text{red}_i}, \overline{\text{red}_i}$ for the red_i-value of this type as follows: We can compute bluefit_i and redfit_i only depending on the upper bound of the size of items of this type, i.e. depending on x, the lower bound of the next larger item size. First, we require $\frac{1-\text{red}_i}{\text{bluefit}_i t_{i+1}} \leq 1$, which gives $\text{red}_i \geq 1 - s_i \cdot \text{bluefit}_i =: \underline{\text{red}_i}$. Second, we want to make sure that the maximum expansion of the current type is not larger than the expansion of the previous (next larger) type (since that might slow down the search), exp_{i-1}: $\frac{1-\text{red}_i}{\text{bluefit}_i s_i} + \frac{\text{red}_i}{\text{redfit}_i s_i} \leq exp_{i-1} \Leftrightarrow \text{red}_i \leq \frac{\text{bluefit}_i \cdot \text{redfit}_i}{\text{bluefit}_i - \text{redfit}_i}(exp_{i-1} s_i - 1/\text{bluefit}_i) =: \overline{\text{red}_i}$. If $\underline{\text{red}_i} \leq \overline{\text{red}_i}$, we continue to test $(1/(j+1), 1/j]$; if not, we know that the *previously tested* type is necessary to ensure the two constraints. Hence, we add this previous type to the list of types, together with the value $\underline{\text{red}_{i-1}}$ computed in the previous iteration.

Computation of redspace**-values** We generated the redspace-values completely automatically, in contrast to Seiden's paper, where these values are defined by the author explicitly. For every type i such that $t_{i+1} \in [1/6, 1/3]$, t_{i+1} is added as a redspace-value and for every type i such that $2 \cdot t_{i+1} \in [1/6, 1/3]$, $2 \cdot t_{i+1}$ is added as a redspace-value. Additionally, we make sure that for each medium type we have a redspace-value equal to x and one equal to $1 - 2x$ where x is the lower bound of the size of items of this type.

After computing the functions leaves and needs, we eliminate redspace-values that are unused and less than 1/3, i.e., if there is no pair of types i, j such that $\text{needs}(i) = \text{leaves}(j) = l, \text{redspace}_l < 1/3$, then redspace_l is removed from the list. This reduces the number of knapsack problems that need to be solved.

Computation and adjustment of values red_i For each item type i that has size at most 1/6 and at least $\mathcal{T}$, we adjust the value red_i such that

$$\frac{1 - \text{red}_i}{\text{bluefit}_i \cdot t_{i+1}} \geq f$$

where $f = 95/100$ if $t_{i+1} \leq 1/13$ and $t_{i+1} > \mathcal{T}$ and $f = 1$ otherwise. To be precise, we set $\text{red}_i = 1 - f \cdot t_{i+1}\text{bluefit}_i$. The reason for this is that it ensures that the "small expansion" of these items, where we count only the blue items of this type, is at least f. This is a heuristic; it does not seem to help to make red_i larger than this.

4.4 Super Harmonic revisited

We revisit the Super Harmonic framework in this section. Seiden used the following weighting functions, but presented them in a different way. Define k and $\mathfrak{r}$ as in definition 14. The two weight functions of an item of type i and size x are defined as follows:

$$w_k(i) = \begin{cases} \beta_i + \rho_i & \text{if } i < N, \text{needs}(i) \geq k \text{ or needs}(i) = 0 \\ \beta_i & \text{if } i < N, 0 < \text{needs}(i) < k \\ \frac{1}{1-\varepsilon} x & \text{if } i = N \end{cases}$$

$$v_k(i) = \begin{cases} \beta_i + \rho_i & \text{if } i < N, \text{leaves}(i) < k \\ \rho_i & \text{if } i < N, \text{leaves}(i) \geq k \\ \frac{1}{1-\varepsilon} x & \text{if } i = N \end{cases}$$

Using these weight functions, he shows that ineq. (4.9) with $c = 1.58889$ holds for Super Harmonic algorithms. Instead of the mathematical program that Seiden considers, we can use $P_w^{k,\text{sml}}$ and its dual $D_w^{k,\text{sml}}$. We use the method described in section 4.2.8 (a binary search for a weighted average of weights) to check for feasibility of the dual linear programs for all values of k, including the cases where $\mathfrak{r}$ is medium. This is a significantly easier method than the one Seiden used, since it is based on solving standard knapsack problems.

A small modification of our computer program can be used to verify Seiden's result. Surprisingly, this shows that Harmonic++ is in fact 1.58880-competitive. In contrast to Seiden's heuristic program, which took 36 hours to prove Harmonic++'s competitive ratio, our program terminates in a few seconds. Of course, this was over fifteen years ago, but we believe the algorithmic improvement explains a significant

part of the speedup. The fast running time of our approach also allowed us to improve upon Harmonic++ within the Super Harmonic framework: Using improved red_i values, we can show a 1.5884-competitive Super Harmonic-algorithm. Furthermore, these values for red_i are much simpler than the ones Seiden used (which were optimized up to precision $1/2 \cdot 10^{-7}$); they can be found in appendix B.

4.5 Lower bound for Extreme Harmonic-algorithms

We prove a lower bound for any Extreme Harmonic algorithm. We will consider inputs consisting of essentially four different item sizes: $1/2+\varepsilon$, $1/3+\varepsilon$, $1/4+\varepsilon$, and $1/7+\varepsilon$ (we also speak of types 1 through 4). Here ε is a very small number. However, there will be many different item sizes in the range $(1/3, 1/3+\varepsilon]$. The value of ε is chosen small enough that the algorithm puts all these sizes in the same type. Note that the algorithm has not much choice about how many red items of types 2 and 3 can be packed in one bin: only one such item can be packed, else larger blue items could not be added anymore. For type 4, between 1 and 3 red items could be packed in one bin, and we will give lower bound constructions for each of these three cases.

Consider the case that the algorithm packs red type 4 items pairwise into bins. In table 4.3, we give four different inputs that together will prove a lower bound of 1.5762 for this case. A pattern (a, b, c, d) denotes a set of items containing a items of type 1, b items of type 2 and so on. Note that our types defined here do not necessarily correspond to size thresholds used by the algorithm; nevertheless, each item gets a single type assigned by the algorithm, and if we use notation such as redfit_i for type i as defined here, we mean the redfit-value of the item type the algorithm assigns to such an item. The other two columns of the table are explained below.

Table 4.3: Inputs for lower bound 1.5762 in case $\mathrm{redfit}_4 = 2$.

Pattern	Space for $\varepsilon \to 0$	Distribution χ
0 0 3 1	$\frac{31}{28} + 2\mathrm{red}_3 + \frac{1-\mathrm{red}_4}{6} + \frac{\mathrm{red}_4}{2}$	1
1 1 0 0	$1 + \frac{1-\mathrm{red}_2}{2} + \frac{1}{6}$	1
1 1 0 1	$1 + \frac{1-\mathrm{red}_2}{2} + \frac{1-\mathrm{red}_4}{6} + \frac{1}{42}$	1
0 2 1 0	$2 \cdot \frac{1+\mathrm{red}_2}{2} + \frac{1-\mathrm{red}_3}{3} + \frac{1}{12}$	1 (scaled)
q^1	$1 + \frac{1-\mathrm{red}_2}{2} + \mathrm{red}_2$	$\frac{2(1-\mathrm{red}_2-\mathrm{red}_2\cdot\mathrm{red}_3)}{1+\mathrm{red}_2}$
q^2	$1 + \frac{1-\mathrm{red}_2}{2} + \mathrm{red}_2$	$2\mathrm{red}_3$

The first three lines of the table represent three different inputs to the algorithm, and the last three lines together represent the final input used in the lower bound. We construct the first three inputs as follows. For each pattern in the table, items arrive in order from small to large. Each item in the pattern arrives N times. In addition, we get N times some amount of sand per bin, that fills up the bin completely. Based on each pattern and the values red_i and redfit_i, we can calculate exactly how much space (represented as fractions of bins) the online algorithm needs to pack each item in the pattern *on average*. To do this, we assume that if red small items can be packed with larger blue ones, the algorithm will always do this (this is a worst-case assumption). The result of this calculation is shown in the column Space.

To illustrate this approach, let us consider an input based on the pattern $(0,0,3,1)$ in the manner described above. As we assumed that $\mathrm{redfit}_4 = 2$, we know that items of types 3 and 4 will not be combined by the algorithm, as $3/4 + 2/7 > 1$. Thus, the algorithm will not be able to combine the red items of both types with any other items. The number of bins used for blue type 3 items is at least $3 \cdot (\frac{1-\mathrm{red}_3}{3})N$, the number of bins for red type 3 items is at least $3 \cdot \mathrm{red}_3 N$. Analogously, we need at least $\frac{1-\mathrm{red}_4}{6}N$ bins for blue type 4 items and at least $\frac{\mathrm{red}_4}{2}N$ bins for red type 4 items. Finally, sand of total volume arbitrarily close to $(1-3/4-1/7)N = \frac{3}{28}N$ arrives, which is packed in at least as many bins by the online algorithm. Thus, on average the items in this pattern need $3 \cdot (\frac{1-\mathrm{red}_3}{3}) + 3 \cdot \mathrm{red}_3 + \frac{1-\mathrm{red}_4}{6} + \frac{\mathrm{red}_4}{2} + \frac{3}{28} = \frac{31}{28} + 2\mathrm{red}_3 + \frac{1-\mathrm{red}_4}{6} + \frac{\mathrm{red}_4}{2}$ bins to be packed. The space needed for the second and third patterns can be calculated in the same way.

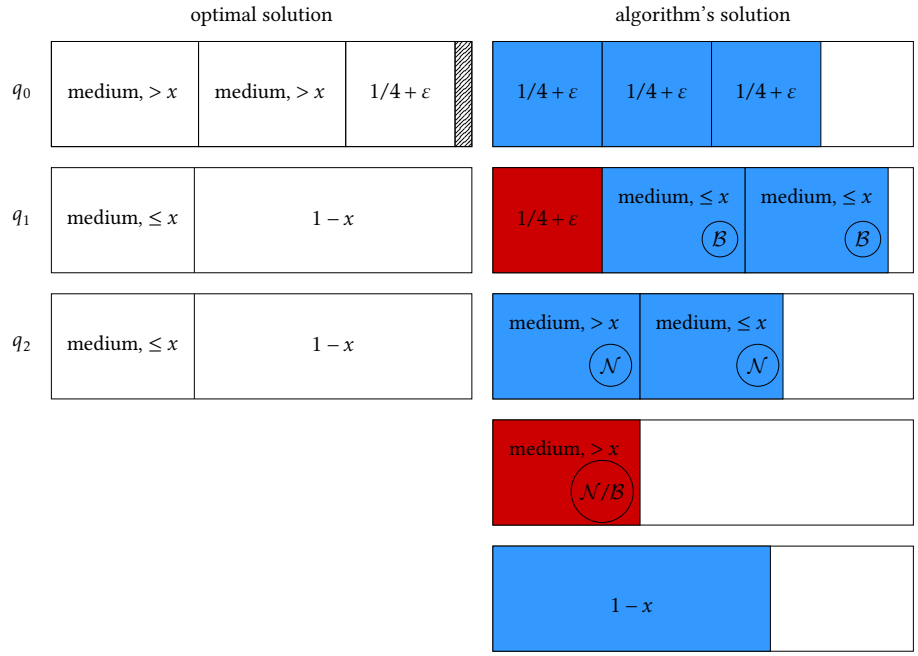


Figure 4.8: Fourth input for our lower bound construction. The three patterns used in the optimal solution are depicted on the left. The shaded area in the first pattern denotes sand. The algorithm produces the five types of bins depicted on the right, plus bins that only contain sand (not depicted here).

The fourth input (based on pattern (0,2,1,0)) requires more explanation; see also fig. 4.8. For this input, we consider a combination of three patterns that arrive in the distribution given in the last column of the table. Items of type 2 have size $1/3 + \varepsilon$ and some of them end up alone in bins. We extend the input in this case by a number of items of size almost $2/3$, where this number is calculated as explained below. All these large items will be placed in new bins by the online algorithm. In order for this to hold, the items of type 2 must have slightly different sizes - not all exactly $1/3 + \varepsilon$. We therefore pick ε small enough so that the interval $(1/3, 1/3 + \varepsilon]$ is contained in a single type according to the classification done by the algorithm. The first item of this type will have size $1/3 + \varepsilon/2$. The sizes of later items depend on how it is packed:

- If the item is packed in a new bin, all future items will be smaller (in the interval $(1/3, 1/3 + \varepsilon/2]$)
- If the item is packed into a bin with an existing item of type 2 or 3, all future items will be larger (in the interval $(1/3 + \varepsilon/2, 1/3 + \varepsilon]$)

We use the same method for all later items of the same type, each time dividing the remaining interval in two equal halves. By induction, it follows that whenever an item is placed in a new bin, all previous items that were packed first into their bins are larger, and all previous items that were packed into existing bins are smaller. Therefore, after all items of this type have arrived, let x be the size of the last item that was placed into a new bin. (Since the algorithm maintains a fixed fraction of red items of type 2, there can be only constantly many items that arrived after this item; we ignore such items.) We have the following.

- All items of size more than x are packed either alone into bins or are the first item in a bin with two medium but no small red items; and
- All items of size less than x are in bins with items of type 3 or were packed as the second item of their type in an existing bin.

We now let items of size exactly $1 - x$ arrive. For every bin with red type 3 items and blue type 2 items, two such items arrive, which will be packed in q^2-bins. Assume that we have N bins with pattern $q^0 = (0, 2, 1, 0)$, then we create exactly $\mathrm{red}_3 N$ such bins, i.e., we let $2\mathrm{red}_3 N$ large items arrive for these. For every bin with a pair of blue medium items but no red items, one such $1 - x$ item arrives. The number of these bins is harder to calculate. Let M be the total number of medium items in the input. Then the number of such bins is $\frac{1-\mathrm{red}_2}{2} M - \mathrm{red}_3 N$. Now, we want to express M in terms of N: Observe that N is half the number of medium items larger than x (as only these end up in q^0-bins). The number of those items is equal to the number of bins with red medium items (which is $\mathrm{red}_2 M$) plus the number of bins with two blue medium but no red items (which is $\frac{1-\mathrm{red}_2}{2} M - \mathrm{red}_3 N$). Thus, N is equal to $\frac{1}{2}\left(\mathrm{red}_2 M + \frac{1-\mathrm{red}_2}{2} M - \mathrm{red}_3 N\right)$. This shows that $M = \frac{4+2\mathrm{red}_3}{1+\mathrm{red}_2} N$. Finally, we conclude that we can send $\frac{1-\mathrm{red}_2}{2} M - \mathrm{red}_3 N = \frac{1-\mathrm{red}_2}{2} \cdot \frac{4+2\mathrm{red}_3}{1+\mathrm{red}_2} N - \mathrm{red}_3 N = \frac{2(1-\mathrm{red}_2-\mathrm{red}_2\cdot\mathrm{red}_3)}{1+\mathrm{red}_2} N$ many large items and thus get this many q^1-bins.

To pack N copies of a given pattern, the online algorithm needs N times the space calculated in table 4.3, while the optimal solution needs exactly N bins. In order to calculate the final lower bound, for each of the four inputs, we simply calculate the space of the pattern(s), in the last case the weighted (in proportion to the distribution) sum of the three patterns' spaces. All four cases inputs a lower bound of at least 1.5762, which is achieved if $\mathrm{red}_1 = 0, \mathrm{red}_2 = 0.1800, \mathrm{red}_3 = 0.1276, \mathrm{red}_4 = 0.1428$. Whenever an algorithm has a smaller or larger value for some red_i value, the space needed by one of the patterns (or the weighted sum of the spaces needed by the three patterns of the last case) increases and thus gives a lower bound above 1.5762.

Constructions for the other two cases $\mathrm{redfit}_4 = 1$ and $\mathrm{redfit}_4 = 3$ can be found below in tables 4.4 and 4.5. The analysis is completely analogous to the first case. For the case $\mathrm{redfit}_4 = 1$, the best values the online algorithm can use are $\mathrm{red}_1 = 0, \mathrm{red}_1 = 0.19, \mathrm{red}_2 = 0.0872$. The analysis for the case $\mathrm{redfit}_4 = 3$ is particularly simple, as the given distribution requires 100/63 bins on average (independent of red_2 and red_3), implying a lower bound of $100/63 \approx 1.5873$.

Table 4.4: Inputs for lower bound 1.5788 in case $\text{redfit}_4 = 1$.

Pattern	Space for $\varepsilon \to 0$	Distribution χ
1 1 1	$1 + \frac{1-\text{red}_2}{2} + \frac{1-\text{red}_3}{6} + \frac{1}{42}$	1
0 0 6	$6 \cdot \frac{1-\text{red}_3}{6} + 6\text{red}_3 + \frac{1}{7}$	1
0 2 2	$2 \cdot \frac{1+\text{red}_2}{2} + 2 \cdot \frac{1-\text{red}_3}{6} + \frac{1}{21}$	1 (scaled)
q^1	$1 + \frac{1-\text{red}_2}{2} + \text{red}_2$	$\frac{4(1-\text{red}_2-\text{red}_2\cdot\text{red}_3)}{1+\text{red}_2}$
q^2	$1 + \frac{1-\text{red}_2}{2} + \text{red}_2$	4red_3

Table 4.5: Inputs for lower bound 1.5872 in case $\text{redfit}_4 = 3$.

Pattern	Space for $\varepsilon \to 0$	Distribution χ
1 1 1	$1 + \frac{1-\text{red}_2}{2} + \frac{1-\text{red}_3}{6} + \frac{1}{42}$	2/3
0 2 2	$2 \cdot \frac{1+\text{red}_2}{2} + 2 \cdot \frac{1+\text{red}_3}{6} + \frac{1}{21}$	1/3

4.6 A further improvement: Introducing red sand

We will now describe how the framework Extreme Harmonic can be improved further by also coloring sand, i.e., items of type N, red and blue. This is also one building block amongst others used in the recent paper by Balogh et al. [10], who achieve a competitive ratio of 1.5783 using a different analysis. It allows us to set some of the parameters differently and we obtain an algorithm with competitive ratio 1.5787.

4.6.1 The algorithm and its properties

In order to incorporate the coloring of sand in the framework, we define an additional parameter red_N that describes the ratio of *red bins* (rather than red items), that is,

$$\text{red}_N \approx (\#\text{ of bins with red sand})/(\#\text{ of bins with red or blue sand}).$$

We will also have $\text{leaves}(N) = 0$. We do not use bluefit- or redfit-parameters for type N, as the number of items of this type in a bin depends on the size of these items.

We now discuss the changes that need to be made to the algorithm. We first extend the definition for *open bins*.

Definition 18 (Extension of definition 9)**.** *A bin is* red-open *for a non-sand item of type t if it contains at least one and at most* $\text{redfit}_t - 1$ *red t items. A bin is* blue-open *for a non-sand item of type t if it contains at least one and at most* $\text{bluefit}_t - 1$ *blue t items.*

A bin is red-open *for a sand item of size u, if it contains red sand items of total size in* $(0, \text{redspace}_{\text{needs}(N)} - u]$*. A bin is* blue-open *for a sand item of size u, if it contains blue sand items of total size in* $(0, 1 - u]$.

Definition 11 simplifies for sand items as follows:

Definition 19. *An unmixed bin is* red-compatible *with a sand item $\mathfrak{p}$ if the bin contains blue or uncolored items of type i and* $\text{leaves}(i) \geq \text{needs}(N)$.

For sand items, blue-compatible bins can never exist, as blue sand items are packed in pure blue bins. This is also reflected in the following algorithm requirements, replacing requirements R(5) and R(6):

R(5') We have $t_1 = 1, t_2 = 2/3, t_3 = 1/2$, and $\text{red}_1 = \text{red}_2 = 0$.

R(6') All type 1 items (i.e., huge items) as well as blue type N items are packed in pure blue bins. Equivalently, $\text{leaves}(1) = \text{leaves}(N) = 0$.

The algorithm used to pack sand items is described in algorithm 11. Non-sand items are packed as before. Note that n^N and n_r^N count the number of *bins* (rather than items, as for n^i and n_r^i for $i < N$). The packing of sand items is basically the same as for non-sand items, the main difference in the algorithm is how the counters n^N and n_r^N are increased.

Algorithm 11 How to pack a single item $\mathfrak{p}$ of type N. At the beginning, we set $n_r^N \leftarrow 0$ and $n^N \leftarrow 0$.

if $n_r^N < \lfloor \text{red}_N n^N \rfloor$ **then**
  if $\exists j : n_{\text{bonus}}^j > 0$ **then**
    // *label the bonus item as sand and color the new sand item blue*
    Let $\mathfrak{b}$ be a bonus item of such a type j
    $n_{\text{bonus}}^{t(\mathfrak{b})} \leftarrow n_{\text{bonus}}^{t(\mathfrak{b})} - 1$
    Label $\mathfrak{b}$ as type N
    $n^N \leftarrow n^N + 1$
    $n_r^N \leftarrow n_r^N + 1$
    $\text{Pack}_{\text{sand}}(\mathfrak{p}, \text{blue})$
  else // *color the new sand item red and pack it*
    $\text{Pack}_{\text{sand}}(\mathfrak{p}, \text{red})$
  end if
else
  $\text{Pack}_{\text{sand}}(\mathfrak{p}, \text{blue})$
end if

Algorithm 12 $\text{Pack}_{\text{sand}}(\mathfrak{p}, c)$ for $c \in \{\text{blue}, \text{red}\}$

Try the following types of bins to place $\mathfrak{p}$ in this order
  • a pure blue, mixed, or unmixed c-open bin with items of type N and color c
  • a c-compatible unmixed bin (the bin becomes mixed, with fixed colors of its items)
  • a new unmixed bin (or pure blue bin, if $c = \text{blue}$)
Give $\mathfrak{p}$ the color c
if $\mathfrak{p}$ is the first sand item in its bin **then**
  $n^N \leftarrow n^N + 1$
  If $c = \text{red}$, then $n_r^N \leftarrow n_r^N + 1$
end if

We know the following, replacing property 1:

Property 1. *Each bin containing blue items of type N, apart from possibly the last one, contains items of total size at least $1-\varepsilon$. Each bin containing red items of type N, apart from possibly the last one, contains red items of total size at least* $\text{redspace}_{\text{needs}(N)} - \varepsilon$.

Properties 2 and 4 and invariant 1 holds for sand items as well. We next show an extension of property 5 for sand items.

Lemma 13. *At all times,* $\lfloor \text{red}_N n^N \rfloor - 1 \leq n_r^N \leq \lfloor \text{red}_N n^N \rfloor + 1$.

Proof. In the beginning, all counters are zero and the bounds hold. We have the lower bound of n_r^N due to line 1 in algorithm 11 and because n^N is increased by at most one in between two consecutive times that this condition is tested. We get the upper bound because n_r^N increases only if $n_r^N < \lfloor \text{red}_N n^N \rfloor$ and a new item of this type arrives (line 1 of algorithm 11), and it increases by at most one (line 8 of algorithm 12). □

All other properties and lemmas are not affected by coloring sand. Most of the packing properties concern only medium items, so they are not affected either. In post-processing, we now remove the single bin with blue items of type N and total size at most $1-\varepsilon$, as well as the single bin with red items of type N and total size at most $\text{redspace}_{\text{needs}(N)} - \varepsilon$ (due to the adapted property 1). This leads to an extended version of packing property 6:

Packing Property 8 (Replacement of packing property 6)**.** *All bins with blue items of type N are at least $1-\varepsilon$ full. All bins with red items of type N are at least* $\text{redspace}_{\text{needs}(N)} - \varepsilon$ *full.*

The change of marks and the coloring of previously uncolored items does not interfere with our new red sand items. Algorithm 7 can be applied the same way as before: Only in lines 9 and 10, if $j = N$, we split up $\mathfrak{p}$ into $x = \lceil s(\mathfrak{p})/t_N \rceil$ items of size $s(\mathfrak{p})/x \leq t_N$ each instead. Lemma 6 does still hold for this case, as the red sand items still occupy space of at least $s(\mathfrak{p}) > 1/3 > \text{redspace}_{\text{needs}(N)}$ (using requirement R(1)). All in all, theorem 1 still holds (the proof uses lemma 13).

4.6.2 Weighting functions

Lemma 8 still holds and we can still define k and $\mathfrak{r}$ as in definition 14. It might now also happen that $k < K+1$ and $t(\mathfrak{r}) = N$, i.e., the item $\mathfrak{r}$ is a sand item.

Recall that $\beta_t = \frac{1-\text{red}_t}{\text{bluefit}_t}$ and $\rho_t = \frac{\text{red}_t}{\text{redfit}_t}$ for $t < N$. Analogously, we now define $\beta_N = \frac{1-\text{red}_N}{1-\varepsilon}$ and $\rho_N = \frac{\text{red}_N}{\text{redspace}_{\text{needs}(N)}-\varepsilon}$. We can now re-define the weighting functions.

$$w_k(\mathfrak{p}) = w_k(x,t,\mathcal{M}) = \begin{cases} \beta_t + \rho_t & \text{if } t < N \text{ and } (\text{needs}(t) > k \text{ or} \\ & \quad \text{needs}(t) = 0 \text{ or} \\ & \quad (\text{needs}(t) = k \text{ and } \mathcal{M} \neq \mathcal{R})) \\ \beta_t & \text{if } t < N \text{ and either } 0 < \text{needs}(t) < k \\ & \text{or needs}(t) = k, \mathcal{M} = \mathcal{R} \\ \beta_N x & \text{if } t = N \text{ and needs}(N) < k \\ (\beta_N + \rho_N)x & \text{if } t = N \text{ and needs}(N) \geq k \end{cases}$$

$$v_{k,s(\mathfrak{r})}(\mathfrak{p}) = v_{k,s(\mathfrak{r})}(x,t) = \begin{cases} \beta_t + \rho_t & \text{if } t < N, \text{leaves}(t) < k \\ \rho_t & \text{if } t < N, \text{leaves}(t) \geq k, k \leq K \\ (\beta_N + \rho_N)x & \text{if } t = N \end{cases}$$

Note that the function $v_{k,s(\mathfrak{r})}$ again depends on $s(\mathfrak{r})$ in case $\mathfrak{r}$ is medium; as before, we set $v_k(\mathfrak{p}) = v_{k,t_{t(\mathfrak{r})}}(\mathfrak{p}) \geq v_{k,s(\mathfrak{r})}$. In the case that $k = \text{needs}(N)$, i.e., the item $\mathfrak{r}$ is a sand item, this function does not depend on the exact size of the item $\mathfrak{r}$. We hence again use $v_k = v_{k,t_{t(\mathfrak{r})}}$. The proof of theorem 2 now runs completely analogous to before. We give a proof sketch which focuses on the changes from the proof of theorem 2.

Theorem 3. *For any input σ and Extreme Harmonic algorithm $\mathcal{A}$, defining k as above we have*

$$\mathcal{A}(\sigma) \leq \min\left\{\sum_{i=1}^{n} w_k(\mathfrak{p}_i), \sum_{i=1}^{n} v_k(\mathfrak{p}_i)\right\} + O(1) \tag{4.28}$$

Proof. We again upper bound $\mathcal{A}(\sigma)$ by the weights of the items $\mathfrak{p}_1, \ldots, \mathfrak{p}_n$, which are the items in σ'. The additive constant $O(1)$ again corresponds to the bins removed in post-processing.

Let Tiny be the total size of the items of type N in σ'. Let UnmixedRed be the number of unmixed red bins in P'. Let B_i and R_i be the number of bins in P' containing blue items of class i, and red items of class i, respectively. In contrast to the proof of theorem 2, we count sand items in B_i as well.

If UnmixedRed $= 0$, every red item is placed in a bin with one or more blue items, and $k = K + 1$. In this case, the total number of bins in P' is exactly the total number of bins containing blue items. Each bin containing blue items of type N contains at least a total size of $1 - \varepsilon$ due to packing property 8 and the blue type N items have total volume $(1 - \text{red}_N)$Tiny. Thus, there are Tiny$(1 - \text{red}_N)/(1 - \varepsilon) = \beta_N$Tiny $= \sum_{i:t(\mathfrak{p}_i)=N} w_{K+1}(\mathfrak{p}_i)$ bins with blue sand items. Note that $v_{K+1}(\mathfrak{p}_i) \geq w_{K+1}(\mathfrak{p}_i)$ for a sand item $\mathfrak{p}_i$.

In bins with blue items of type $t < N$, exactly bluefit_t such items are packed according to packing property 5. Thus, we need $\frac{(1-\text{red}_t)n_t}{\text{bluefit}_t}$ bins to pack the blue items of type t, where n_t is the total number of items of type t. For each item $\mathfrak{p}$ of type $t < N$, we have $w_{K+1}(\mathfrak{p}) = \beta_t = \frac{1-\text{red}_t}{\text{bluefit}_t} \leq v_{K+1}(\mathfrak{p})$. We see that w_{K+1} counts all the bins with blue items, and

$$\mathcal{A}(\sigma) \leq \sum_{i=0}^{K} B_i \leq \sum_{i=1}^{n} w_{K+1}(\mathfrak{p}_i).$$

If UnmixedRed > 0, then $k = \text{needs}(t(\mathfrak{r}))$, and there is an unmixed red bin of class k. As before, we get that the total number of bins in P' is at most

$$\begin{aligned} &\textsc{UnmixedRed} + \sum_{i=0}^{K} B_i + O(1) \\ \leq\ & B_0 + \min\left\{ \sum_{i=k+1}^{K} R_i + R_k(-\mathcal{R}) + \sum_{i=1}^{K} B_i, \sum_{i=1}^{K} R_i + \sum_{i=1}^{k-1} B_i \right\} + O(1). \end{aligned} \tag{4.29}$$

Let J be the set of types whose blue items are packed in pure blue bins, including type 1 and type N. For each item $\mathfrak{p}$ of type $t \neq N$, $t \in J$, we have $\text{leaves}(t) = 0 < k$, so $v_k(\mathfrak{p}) = \frac{1-\text{red}_t}{\text{bluefit}_t}$. Furthermore, $w_k(\mathfrak{p}) \geq \frac{1-\text{red}_t}{\text{bluefit}_t}$. For an item $\mathfrak{p}$ of type $t = N$, we have $v_k = (\beta_N + \rho_N)s(\mathfrak{p})$ and $w_k \geq \beta_N s(\mathfrak{p})$. We conclude $\sum_{j\in J}\sum_{t(\mathfrak{p}_i)=j} w_k(\mathfrak{p}_i) \geq \sum_{j\in J}\sum_{t(\mathfrak{p}_i)=j} v_k(\mathfrak{p}_i) \geq B_0$ using packing property 5.

In the first term of the minimum in ineq. (4.29), we count all bins with blue items except the pure blue bins, all bins with red items of classes above k, and the bins with red items of class k that are not marked $\mathcal{R}$. (If red items of class k are small, this means all red items of this class.) This term is therefore upper bounded by $\sum_{j\notin J}\sum_{t(\mathfrak{p}_i)=j} w_k(\mathfrak{p}_i)$ (again using packing property 5). In the second term of the minimum in ineq. (4.7), we count all bins with red items, as well as bins with blue items of class at least 1 and at most $k-1$. The second term is therefore upper bounded by $\sum_{j\notin J}\sum_{t(\mathfrak{p}_i)=j} v_k(\mathfrak{p}_i)$. □

4.6.3 Offline solution and results

For finding the lower bound on the optimal cost depending on the value of k, we can apply the same steps as described in section 4.2.7. We do not need to consider the case that $\mathfrak{r}$ is a sand item separately, as this is simply one particular value for k and we check all such values. We can define our types the same way as before and also the definition of patterns stays unchanged. We still fill up the empty space in a pattern with sand; note that the weight of sand might now be smaller than before, but filling the empty space with sand still increases the total weight of the pattern (as opposed to leaving this space empty). The crucial lemmas 9 and 11 are not affected by the introduction of red sand. However, lemma 12 does not hold anymore, as the w- and v-weights of sand are different. We can, however, prove the following:

Lemma 14. $v_{1k} = v_{2k} \geq w_{1k} = w_{2k}$.

Proof. The sum of the weights of the large and the medium item are, as before, $1 + \beta_{t(N)} + \rho_{t(N)}$. The amount of sand is in both patterns $t_{t(\mathfrak{r})} - t_{t(\mathfrak{r})-1}$. The sand expansion, i.e., the weight of a sand item divided by its size, is at least β_N for w and $\beta_N + \rho_N$ for v. Thus, the v-weights for q^1 and q^2 are equal and not smaller than the w-weights of these patterns. □

Therefore, for simplicity we let $w_{1k} := v_{1k}, w_{2k} := v_{2k}$, i.e., we assume a higher weight for q^1 and q^2 than their actual weight. That way, our bounds still hold and we can still use constraint (4.12) in the dual LP. The only adaption that we now need to make is using the right sand expansion in the program. We adapt definition 17:

Definition 20 (Replacement of definition 17)**.** *In case $D_w^{k,med}$ is used, we define*

$$\omega_k(\mathfrak{p}) = \begin{cases} (1-y_3^*)\left(\beta_{t(\mathfrak{r})} + \rho_{t(\mathfrak{r})}\right) + y_3^* v_k(\mathfrak{p}) + \frac{1-\mathrm{red}_{t(\mathfrak{r})}}{1+\mathrm{red}_{t(\mathfrak{r})}} y_1^* & \text{if } t(\mathfrak{p}) = t(\mathfrak{r}) \\ (1-y_3^*) w_k(\mathfrak{p}) + y_3^* v_k(\mathfrak{p}) + \rho_{t(\mathfrak{p})} y_2^* & \text{if } t(\mathfrak{p}) \in \mathrm{RedComp}(t(\mathfrak{r})) \\ (1-y_3^*) w_k(\mathfrak{p}) + y_3^* v_k(\mathfrak{p}) & \text{else} \end{cases}$$

We now only need to use the sand expansion according to this weighting function in the knapsack problems.

It remains to find new parameters for an algorithm and compute the competitive ratio. There is one crucial change in the generation of types w.r.t. the processes described in section 4.3, namely, adding type threshold of the form $1/2 - x$ for every medium threshold x. This splits up the interval $(1/15, 1/6)$ into further types. Furthermore, to speed up the computation, we reduce the number of tiny items as follows. Instead of inserting a new tiny threshold whenever the small expansion of items becomes too large, we can insert thresholds at multiplicative positions, based on the parameter lastTypeBeforeSmallTypeGeneration $= 1/x$. That is, we multiply the current threshold by $x/(x+1)$ each time and round to get the next threshold of the form $1/a$. Finally, we hand-tune the red-values more carefully. We achieve a competitive ratio of 1.5787 using the parameters specified in appendix C.

4.7 Discussion and directions for future work

We have presented two new algorithms for online bin packing in one dimension, breaking the long standing lower bound by Ramanan et al. [95]. Additionally, we gave a lower bound for our new framework.

The Harmonic-based algorithms that have been studied in the past operate in a rather controlled but local way in the sense that they base their decisions on the compatibility of item types only and do not try to take the bigger picture into account. This is done because it allows for a rather straightforward analysis using weighting functions. Intuition tell us of course that it will make much more sense to adapt our packing depending on the sequence of items seen so far, but the challenge is to incorporate such more adaptive algorithms into the classical analytical frameworks.

In essence, our approach shows that we can leverage information about existing items in order to improve the packing decision for the next item and thus our algorithm's performance overall. The core idea used for our improvement was to combine medium and large items whenever they fit together. If we however just add this to the previous algorithm, the adversary could simply avoid situations where we could make use of this. In order to keep the adversary from doing so, we required two tools:

- By postponing the coloring of items, we try to gain knowledge about the sizes of other items of this type that have arrived up to now. Items marked $\mathcal{N}$ guarantee us this knowledge. On the other hand, if we do not mark medium items with $\mathcal{N}$, we instead know that some other compatible item types arrived earlier and can in turn exploit this knowledge. We thus use the marking to keep track of certain structures in the input that are advantageous for us in different ways, making it possible to bound the number of critical bins (lemmas 9 and 11).

- We relax the assumption about fixed fractions of red items by creating bonus items, which in fact increase the fraction of red medium items beyond the fixed

value. We do this by taking advantage of the fact that we *know* that matching large items exist and can be combined with medium items in this situation, allowing for a good packing of these items.

It seems plausible that future work should try to incorporate the knowledge about existing items even further to obtain better algorithms. One could for example try to relax the assumption about fixed fractions of red items even more. We have taken care that additional red items are only created when this is desirable because they can be combined with large items, but it might be possible that other advantageous situations exist, where increasing or also decreasing of the fraction of red items makes sense. It would be very interesting to develop techniques to generalize this input-dependent adjustment of these parameters to all item types and further scenarios and develop analytical tools to study such algorithms.

Of course, adding more and more constraints increases the complexity of solving the linear program that upper bounds the competitive ratio. In order to overcome these difficulties, we made use of the dual LP and showed how to simplify it by removing variables so that we could solve it using a binary search. If future approaches introduce even more constraints, they will need to develop better techniques for solving the LP in order to make use of increased structural knowledge.

5 Lower Bounds in Multiple Dimensions

In this chapter, we consider the problem of geometric online bin packing in two or more dimensions. We also consider the special case where all items are hypercubes with the same edge length s_i in all dimensions.

Results in this chapter and organization We improve the general lower bound for square packing in two dimensions to 1.680783. Very recently, Epstein et al. improved this lower bound further to 1.75, using different methods [7]. For rectangle packing, we improve the general lower bound to 1.859. Furthermore, we improve the lower bound for Harmonic-type algorithms for hypercube packing in any dimension $d \geq 2$. For this, we generalize the method of Ramanan et al. [95]. In particular, we show that Harmonic-type algorithms cannot break the barrier of 2 for $d = 2$, by giving a lower bound of 2.02 for this case. This shows that substantially new ideas will be needed in order to achieve significant improvements over the current best upper bound of 2.1187 and get closer to the general lower bound. For high dimensions, Our lower bound tends to 3.

Lastly, we show that when additionally incorporating the two central ideas from the one-dimensional bin packing algorithm presented in chapter 4 into two-dimensional square packing, there are instances which pose similarly strong lower bounds as those for Harmonic-type algorithms. This shows that further new ideas are required to improve algorithms for the square packing problem.

Preliminaries At several points in this chapter, we use the notion of *anchor points* as defined by Epstein and van Stee [42]. We assign the coordinate $(0,\dots,0)$ to one corner of the bin, all edges connected to this corner are along a positive axis and have length 1.

Definition 21 (Anchor point, anchor packing)**.** *We say that an item* is placed at an anchor point *if it is placed parallel to the axes such that one of its corners coincides with the anchor point and no point inside the item has a smaller coordinate than the corresponding coordinate of the anchor point. We call an anchor point* blocked *for a certain set of items in a certain packing (i.e. in a bin that contains some items), if we cannot place an item of this set at that anchor point without overlapping other items. See fig. 5.1 for illustration. An* anchor packing *is a packing where every item is placed at an anchor point.*

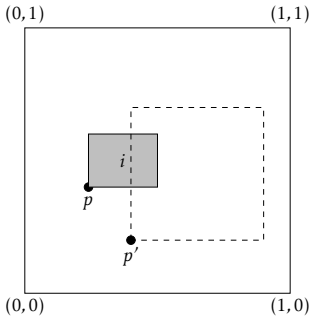


Figure 5.1: Item i is packed at anchor point p. Anchor point p' is blocked for a square item of side length 1/2 (depicted as dashed square) because packing it at this anchor point would cause a collision with item i.

5.1 Lower bound for general algorithms for square packing

5.1.1 Van Vliet's Method

For deriving a general lower bound on the competitive ratio of online hypercube packing algorithms, we extend an approach by van Vliet [107] based on linear programming. Problem instances considered in this approach are characterized by a list of items $L = L_1 \dots L_k$ for some $k \geq 2$, where each sublist L_j contains $\alpha_j \cdot n$ items of side length s_j (we will also call such items "items of size s_j" or simply "s_j-items"). We assume $s_1 \leq \dots \leq s_k$. The input might stop after some sublist. An online algorithm $\mathcal{A}$ does not know beforehand at which point the input sequence stops, and hence the asymptotic competitive ratio can be lower bounded by

$$R \geq \min_{\mathcal{A}} \max_{j=1,\dots,k} \limsup_{n\to\infty} \frac{\mathcal{A}(L_1,\dots,L_j)}{\mathrm{OPT}(L_1,\dots,L_j)}$$

For this approach, we define the notion of a *pattern*[1]: A pattern is a multiset of items that fits in one bin. We denote a pattern by a tuple $(p_1,\dots,p_k)$, where p_i denotes the number of s_i-items contained in the pattern (possibly zero). The performance of an online algorithm on the problem instances we consider can be characterized by the number of bins it packs according to a certain pattern. Van Vliet denotes the set of all feasible patterns by T, which is the union of the disjoint sets $T_1,\dots,T_k$ where T_j contains patterns whose first non-zero component is j (i.e., whose smallest item size used is s_j). We can then calculate the cost of an algorithm $\mathcal{A}$ by $\mathcal{A}(L_1,\dots,L_j) = \sum_{i=1}^{j} \sum_{p\in T_i} n(p)$, where $n(p)$ denotes the number of bins $\mathcal{A}$ packs according to pattern p. We call a pattern $p \in T_j$ *dominant* if the multiset consisting of the items of p plus one s_j-item cannot be packed in one bin. Note that we only need to consider dominant patterns in the LP [107]; thus we from now on denote by T_j the set of all *dominant* feasible patterns with no items smaller than s_j. As the variables $n(p)$ characterize algorithm $\mathcal{A}$, optimizing over these variables allows us to minimize the competitive ratio over all online algorithms with the following LP:

[1] This is of course very similar to the patterns defined in section 4.2.7.

sublist L_i	number of items α_i	item size s_i	$\mathrm{OPT}(L_1 \dots L_i) \cdot \frac{176400}{n}$
L_1	$839n$	$1/420 - \varepsilon$	839
L_2	$10n$	$1/105 + \varepsilon/105$	999
L_3	$8n$	$1/84 + \varepsilon/84$	1199
L_4	$4n$	$1/42 + \varepsilon/42$	1599
L_5	$39n$	$1/21 + \varepsilon/21$	17199
L_6	$8n$	$1/20 + \varepsilon/20$	20727
L_7	$4n$	$1/10 + \varepsilon/10$	27783
L_8	$7n$	$1/5 + \varepsilon/5$	77175
L_9	$5n$	$1/4 + \varepsilon/4$	132300
L_{10}	n	$1/2 + \varepsilon/2$	176400

Table 5.1: The input sequence that gives a lower bound of 1.680783 together with optimal solutions.

$$
\begin{array}{lll}
\text{minimize} & R & \\
\text{subject to} & \displaystyle\sum_{p \in T} p_j \cdot x(p) \geq \alpha_j & 1 \leq j \leq k \\
& \displaystyle\sum_{i=1}^{j} \sum_{p \in T_i} x(p) \leq \lim_{n \to \infty} \frac{\mathrm{OPT}(L_1, \dots, L_j)}{n} R & 1 \leq j \leq k \\
& x(p) \geq 0 & \forall p \in T
\end{array}
$$

In this LP, the variables $x(p)$ replace $n(p)/n$, as we are only interested in results for $n \to \infty$. Note that item sizes are always given in nondecreasing order to the algorithm. In this paper, however, we will often consider item sizes in *nonincreasing* order for constructing the input sequence and generating all patterns.

5.1.2 Proving a lower bound of 1.680783

In this section, we will prove the following theorem:

Theorem 4. *No online algorithm can achieve a competitive ratio of* 1.680783 *for the online square packing problem.*

Consider the input sequence in table 5.1. First of all, we need to prove the correctness of the values $\frac{\mathrm{OPT}(L_1 \dots L_j)}{n}$ for $j = 1, \dots, k$ and $n \to \infty$. To prove a lower bound, we do not need to prove optimality of the offline packings that we use. It is sufficient to prove feasibility. To do this, we use anchor packings. In this section, we use 420^2 anchor points. The anchor points are at the positions for which both coordinates are integer multiples of $(1 + \varepsilon)/420$. Note that every item used in the construction apart from the ones in L_1 have sides which are exact multiples of $(1 + \varepsilon)/420$. Therefore, whenever we place an item at an anchor point, and the item is completely contained within the bin, it will fill exactly a square bounded by anchor points on all sides.

To check whether a given pattern is feasible, the items of size s_1 can be considered separately. Having placed all other items at anchor points, we can place exactly one

item of size s_1 at each anchor point which is still available. Here an anchor point (x, y) is available if no item covers the point $(x + \varepsilon, y + \varepsilon)$. By the above, after all other items have been placed at anchor points, it is trivial to calculate the number of available anchor points; at least all the anchor points with at least one coordinate equal to $(1 + \varepsilon)419/420$ are still available.

For any pattern that we use, the largest items in it are always arranged in a square grid at the left bottom corner of the bin (at anchor points). The second largest items are arranged in an L-shape around that square. It is straightforward to calculate the numbers of these items as well. The patterns used for the given upper bounds on the optimal solution are listed in table 5.2. Note that not all of these patterns are greedy (in the sense that we add, from larger to smaller items, always as many items of the current type as still fit).

Let us give some intuition on how these patterns are constructed. We start by finding a pattern that contains the maximal number of the largest type of items, and then add greedily as many items as possible of the second largest type, then third largest type and so on. We take as many bins with this pattern as are necessary to pack all the largest items; a certain number of items of all other types remain. We continue by choosing the pattern that contains the largest possible number of items of the second-largest type and fill it up greedily as before with other items. We use this pattern in such a number of bins that all remaining items of the second-largest type are packed. We continue like that until all items are packed. You can see this approach for example in the patterns used for $\mathrm{OPT}(L_1 \dots L_3)$. We can pack 6889 items of size s_3 into one bin. With these, we can pack no more than 207 s_2-items, and finally we can add at most 863 s_1-items; this gives the first pattern, $p^{(3)}$. We need $n \cdot 8/6889$ bins with this pattern to pack the $8n$ s_3-items. This leaves $n \cdot 67234/6889$ items of size s_2 unpacked, and as we can pack at most 10816 s_2-items into one bin (and 3344 s_1-items with them), this gives a certain amount of bins with this second pattern $p^{(2)}$. For the remaining s_3-items, we then use the respective number of bins with pattern $p^{(1)}$.

However, we sometimes slightly derive from this construction, e.g., in the patterns used for $\mathrm{OPT}(L_1 \dots L_5)$. In a bin with 400 items of size s_5, we could fit 81 items of size s_4. However, if we do so, we would pack more s_4-items than necessary and thus lose space that we need in order to pack other items. In that case, we reduce the number of s_4-items as much as possible while still packing all of them (in this case, we reduce it to 42). In fig. 5.2, we give the optimal packing for the whole input sequence (i.e., for $L_1 \dots L_{10}$).

In order to prove lower bounds, we will use the dual of the LP given above. It is defined as follows:

$$
\begin{array}{lll}
\text{maximize} & \sum_{j=1}^{k} \alpha_j \lambda_j & \\
\text{subject to} & \displaystyle\sum_{i=j}^{k} \lambda_i p_i + \sum_{i=j}^{k} \mu_i \leq 0 & \forall p \in T_j, 1 \leq j \leq k \\
& \displaystyle -\sum_{j=1}^{k} \mu_j \cdot \lim_{n \to \infty} \frac{\mathrm{OPT}(L_1 \dots L_j)}{n} \leq 1 & \\
& \lambda_j \geq 0 & 1 \leq j \leq k \\
& \mu_j \leq 0 & 1 \leq j \leq k
\end{array}
$$

Note that any feasible solution to this dual gives us a valid lower bound for the

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}
$p^{(1)}$	176400									
$p^{(2)}$	3344	10816								
$p^{(3)}$	863	207	6889							
$p^{(4)}$	863	207	165	1681						
$p^{(5)}$	10477	103	83	42	400					
$p^{(6)}$	839	16	8	4	39	361				
$p^{(7)}$	10493	102	83	42	400					
$p^{(8)}$	839	16	8	4	39	37	81			
$p^{(9)}$	10541	99	83	42	400					
$p^{(10)}$	1918	23	18	10	90	19	10	16		
$p^{(11)}$	839	10	8	4	39	8	4	7	9	
$p^{(12)}$	1918	23	19	10	90	19	10	16		
$p^{(13)}$	839	10	8	4	39	8	4	7	5	1

i	patterns for OPT_i	number of bins with this pattern divided by n
1	$p^{(1)}$	$\frac{839}{176400}$
2	$p^{(2)}$	$\frac{10}{10816}$
	$p^{(1)}$	$\frac{31393}{6624800}$
3	$p^{(3)}$	$\frac{8}{6889}$
	$p^{(2)}$	$\frac{33617}{37255712}$
	$p^{(1)}$	$\frac{1944236893}{410744224800}$
4	$p^{(4)}$	$\frac{4}{1681}$
	$p^{(3)}$	$\frac{12788}{11580409}$
	$p^{(2)}$	$\frac{31961}{37255712}$
	$p^{(1)}$	$\frac{646638631}{136914741600}$

i	patterns for OPT_i	number of bins with this pattern divided by n
5	$p^{(5)}$	$\frac{39}{400}$
6	$p^{(6)}$	$\frac{8}{361}$
	$p^{(7)}$	$\frac{13767}{144400}$
7	$p^{(8)}$	$\frac{4}{81}$
	$p^{(6)}$	$\frac{500}{29241}$
	$p^{(9)}$	$\frac{13143}{144400}$
8	$p^{(10)}$	$\frac{7}{16}$
9	$p^{(11)}$	$\frac{5}{9}$
	$p^{(12)}$	$\frac{7}{36}$
10	$p^{(13)}$	1

Table 5.2: Patterns used for the optimal solutions. The upper table describes the patterns used. The lower tables describe which patterns are used in which cases and for how many bins; we use OPT_i as a shortcut for $\mathrm{OPT}(L_1 \dots L_i)$.

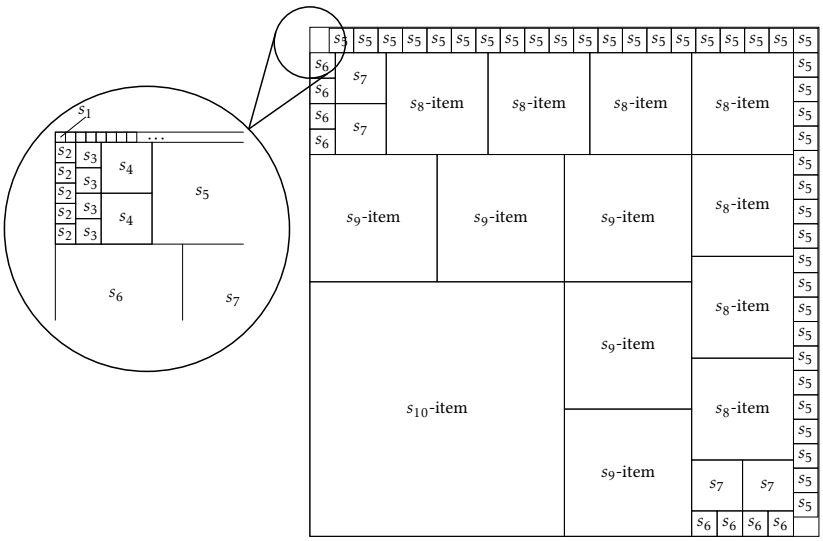


Figure 5.2: How to pack pattern $p^{(13)}$ from table 5.2. Note that the sketch is not true to scale for the sake of readability.

i	λ_i/x	$-\mu_i/x$
1	1	863
2	16	3312
3	25	4125
4	100	8100
5	400	15600

i	λ_i/x	$-\mu_i/x$
6	400	14800
7	1600	27200
8	6400	44800
9	6400	32000
10	25600	25600

Table 5.3: The variable values for the dual solution. Here, $x = 4410/338989303$.

problem. In table 5.3, we specify a solution and then prove that it is indeed feasible for the dual LP. In this table, the constant x is defined as $4410/338989303$.

Note that the dual constraint

$$-\sum_{j=1}^{k} \mu_j \cdot \lim_{n\to\infty} \frac{\mathrm{OPT}(L_1 \dots L_j)}{n} \leq 1$$

is satisfied with equality.

It thus remains to check the other constraints, where we have one constraint for every pattern. For verifying that all constraints $\sum_{i=j}^{k} \lambda_i p_i + \sum_{i=j}^{k} \mu_i \leq 0$ are satisfied, we see that it suffices to check that for every $j = 1, \dots, k$ the inequality $\max_{p \in T_j} \sum_{i=j}^{k} \lambda_i p_i \leq -\sum_{i=j}^{k} \mu_i$ holds. We can interpret the λ_i values as weights assigned to items of type i, let $w(p) = \sum_{i=j}^{k} \lambda_i p_i$ for $p \in T_j$, and thus the problem reduces to finding the pattern in T_j with maximum weight – a knapsack problem. The μ_i-values define the capacity of the knapsack. In order to solve this efficiently, we introduce a dominance notion for items.

Definition 22. *We say that m^2 items of size s_i dominate an item of size s_j, denoted by $m^2 s_i \succ s_j$, if $m s_i \leq s_j$ and $m^2 \lambda_i \geq \lambda_j$.*

In the case that $m^2 s_i$-items dominate an s_j-item, we can replace one item of size s_j by m^2 items of size s_i (arranged in an $m \times m$ grid), as the items to do not take more space. Furthermore, the weight of the pattern only increases by this replacement step, so it suffices to only examine the pattern without s_j items. Note that the $\succ$-operator is transitive.

For our input, we use the following dominance relations:

$$4^2 s_1 \succ s_2 \qquad 5^2 s_1 \succ s_3 \qquad 2^2 s_3 \succ s_4$$
$$2^2 s_4 \succ s_5 \qquad s_5 \succ s_6 \qquad 2^2 s_6 \succ s_7$$
$$2^2 s_7 \succ s_8 \qquad s_8 \succ s_9 \qquad 2^2 s_9 \succ s_{10}$$

It is easy to check that these are indeed fulfilled by the λ_i-values given above. The dominance relations give us that whenever a pattern contains s_1-items, we can replace all other items in this pattern by s_1-items as well – thus, for set T_1, we only need to consider the pattern that contains only s_1-items (and the maximal number of them, i.e., 176400 such items). So using the dominance relation, we have reduced the number of patterns dramatically. Similarly, for $T_3, \dots, T_{10}$, we only have to consider one pattern each. Only for T_2, we have to be careful: As s_3-items are not dominated by s_2 items, we also have to consider patterns that contain s_2 and s_3-items. The following Lemma will show that the pattern that contains 6889 s_3-items and 207 s_2-items is the maximum weight pattern for this case.

Lemma 15. *Among all patterns that only contain items of sizes s_2 and s_3, the pattern with 6889 s_3-items and 207 s_2-items has the highest weight given the λ_i-values of table 5.3.*

Proof. Let p^* be the pattern under consideration. We note that $\frac{\lambda_2}{(1/105)^2} < \frac{\lambda_3}{(1/84)^2}$, i.e., the weight per area is larger for the s_3-items than for the s_2-items. Furthermore, p^* occupies an area of $\left(\frac{104}{105} \cdot (1+\varepsilon)\right)^2$. Note that no pattern with these two item types can cover a larger area. Hence, no pattern can achieve a larger weight. □

In table 5.4, we list all patterns that need to be checked, together with their weight and the knapsack capacity.

Finally, in order to determine the lower bound proven by this input, we compute $839\lambda_1 + 10\lambda_2 + 8\lambda_3 + 4\lambda_4 + 39\lambda_5 + 8\lambda_6 + 4\lambda_7 + 7\lambda_8 + 5\lambda_9 + \lambda_{10} > 1.680783$. This concludes the proof of theorem 4.

5.2 Lower bounds for general algorithms for rectangle packing

We will now present a lower bound on the more general two-dimensional online bin packing, where items are allowed to be arbitrary rectangles and not necessarily squares. In this setting, we receive a sequence of n items, where the i-th item has width w_i and height h_i. The bins are still squares of side length one, and we are not allowed to rotate the items. Note that the LP and its dual are still the same, however, we need to adapt our definition of item dominance as follows.

Definition 23. *We say that $m_1 \times m_2$ items of size s_i dominate an item of size s_j, denoted by $(m_1 \times m_2)s_i \succ s_j$, if $m_1 w_i \leq w_j, m_2 h_i \leq h_j$ and $m_1 m_2 \lambda_i \geq \lambda_j$. Instead of $(1 \times 1)s_i \succ s_j$, we will simply write $s_i \succ s_j$.*

j	heaviest pattern p from T_j	$w(p)/x$	knapsack capacity: $(\sum_{i=j}^{10} -\mu_i)/x$
1	$176400 \times s_1$	176400	176400
2	$207 \times s_2, 6889 \times s_3$	175537	175537
3	$6889 \times s_3$	172225	172225
4	$1681 \times s_4$	168100	168100
5	$400 \times s_5$	160000	160000
6	$361 \times s_6$	144400	144400
7	$81 \times s_7$	129600	129600
8	$16 \times s_8$	102400	102400
9	$9 \times s_9$	57600	57600
10	$1 \times s_{10}$	25600	25600

Table 5.4: The patterns that have to be considered to verify the first set of constraints in the dual LP. Again, $x = 4410/338989303$.

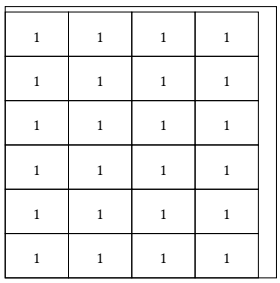
j	w_j	h_j	$\frac{\mathrm{OPT}(L_1 \ldots L_j)}{n}$
		Level 1	
1	$\frac{1}{4} - 300\delta$	$1/6 - 2\varepsilon$	1/24
2	$\frac{1}{4} + 100\delta$	$1/6 - 2\varepsilon$	1/12
3	$\frac{1}{4} + 200\delta$	$1/6 - 2\varepsilon$	1/6
		Level 2	
4	$\frac{1}{4} - 30\delta$	$1/3 + \varepsilon$	1/4
5	$\frac{1}{4} + 10\delta$	$1/3 + \varepsilon$	1/3
6	$\frac{1}{4} + 20\delta$	$1/3 + \varepsilon$	1/2
		Level 3	
7	$\frac{1}{4} - 3\delta$	$1/2 + \varepsilon$	5/8
8	$\frac{1}{4} + 1\delta$	$1/2 + \varepsilon$	3/4
9	$\frac{1}{4} + 2\delta$	$1/2 + \varepsilon$	1

Table 5.5: The input sequence for the 1.859 lower bound. The table shows the items for the first, second and third level. ε and δ are assumed to be sufficiently small positive constants.

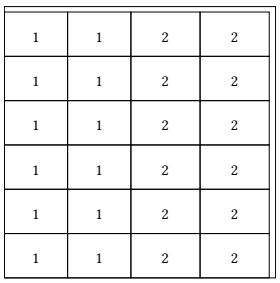
Again, this means that we can replace one item of size s_j by $m_1 m_2$ items of size s_i which are arranged in an $m_1 \times m_2$-grid, while only increasing the weight (sum of λ-values) of the pattern.

5.2.1 A lower bound of 1.859 using nine item types

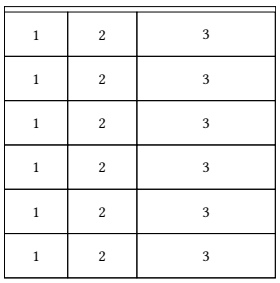
The construction for our lower bound relies on nine item types that are arranged in 3 groups, also called *levels*. Corresponding to these item types, we have nine lists $L_1, \ldots, L_9$, where each L_j consists of n items of size s_j. The item sizes are given in table 5.5 together with the optimal solution values. The packings for the optimal solution are depicted in fig. 5.3.



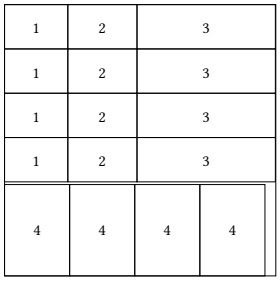
(a) OPT(L_1)



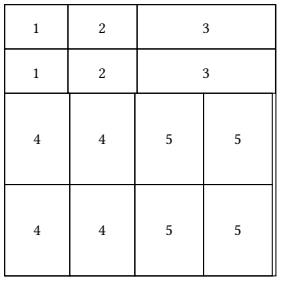
(b) OPT(L_1L_2)

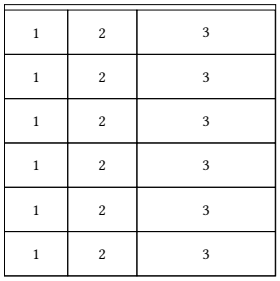


(c) OPT($L_1L_2L_3$)

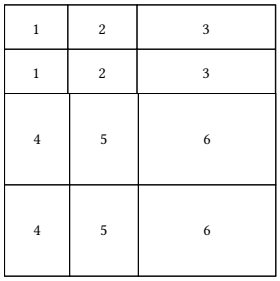


(d) OPT($L_1 \dots L_4$)

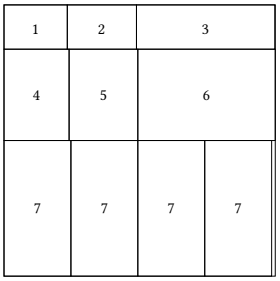


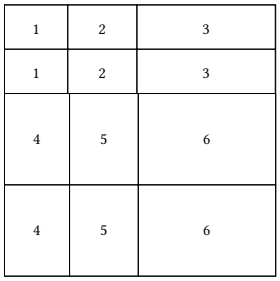


(e) OPT($L_1 \dots L_5$): We have $n/4$ bins with the left packing and $n/12$ bins with the right packing

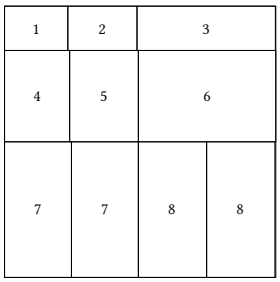


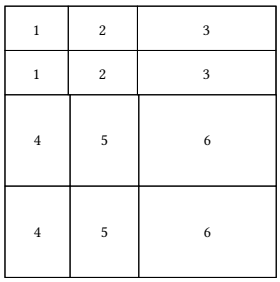
(f) OPT($L_1 \dots L_6$)



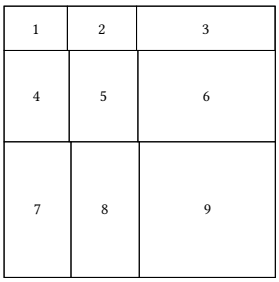


(g) OPT($L_1 \dots L_7$): We have $n/4$ bins with the left packing and $3/8 \cdot n$ bins with the right packing





(h) OPT($L_1 \dots L_8$): We have $n/2$ bins with the left packing and $n/4$ bins with the right packing

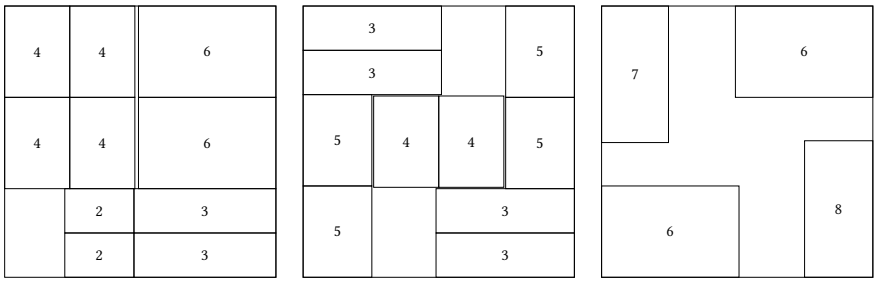


(i) OPT($L_1 \dots L_9$)

Figure 5.3: Optimal solutions for sublists $L_1 \dots L_j$ for all $j = 1, \dots, 9$. The number within every item denotes its type.

pattern	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	type	$x(p)$
$p^{(1)}$	24	-	-	-	-	-	-	-	-	T_1	29/4956
$p^{(2)}$	12	12	-	-	-	-	-	-	-	T_1	289/4956
$p^{(3)}$	12	-	6	-	-	-	-	-	-	T_1	11/826
$p^{(4)}$	-	6	6	-	-	-	-	-	-	T_2	15/413
$p^{(5)}$	-	2	2	4	4	-	-	-	-	T_2	17/1239
$p^{(6)}$	-	2	2	4	-	2	-	-	-	T_2	34/1239
$p^{(7)}$	-	-	4	2	4	-	-	-	-	T_3	64/413
$p^{(8)}$	-	-	-	8	-	-	-	-	-	T_4	55/1239
$p^{(9)}$	-	-	-	2	2	2	-	-	-	T_4	74/1239
$p^{(10)}$	-	-	-	1	1	1	4	-	-	T_4	21/413
$p^{(11)}$	-	-	-	-	1	1	2	2	-	T_5	32/413
$p^{(12)}$	-	-	-	-	1	1	4	-	-	T_5	3/413
$p^{(13)}$	-	-	-	-	1	1	1	1	1	T_5	29/413
$p^{(14)}$	-	-	-	-	-	2	1	1	-	T_6	128/413
$p^{(15)}$	-	-	-	-	-	-	1	1	1	T_7	96/413
$p^{(16)}$	-	-	-	-	-	-	-	1	1	T_8	96/413
$p^{(17)}$	-	-	-	-	-	-	-	-	1	T_9	192/413

Table 5.6: The optimal primal solution for the 1.859 lower bound. The table gives the patterns, the set T_j they belong to, and the value of the LP variable $x(p)$ for each pattern.



(a) $p^{(6)}$ (b) $p^{(7)}$ (c) $p^{(14)}$

Figure 5.4: Packings of some of the patterns of the optimal LP solution from table 5.6. The number within each item denotes its type.

We will now give an optimal solution for the primal LP and then verify its feasibility by checking the constraints, and verify its optimality by giving a matching dual solution and a proof of its feasibility. The optimal primal solution uses 15 different patterns as listed in table 5.6. The packings for some of these (where it is not that easy to see how to pack the pattern) are depicted in fig. 5.4.

To verify feasibility of this solution, note that all the constraints hold with equality (except for the non-negativity constraints of course) if we set $R = 768/413 > 1.859$. For proving optimality, we use the dual solution given in table 5.7. In order to verify

j	$\lambda_j \cdot 413$	$-\mu_j \cdot 413$	types to consider	heaviest patterns p from T_j	$w(p) \cdot 413 = (\sum_{i=j}^{9} -\mu_i) \cdot 413$
1	48	288	1	$24 \times s_1$	1152
2	48	48	2, 4	$18 \times s_2$ $6 \times s_2, 8 \times s_4$ $12 \times s_2, 4 \times s_4$	864
3	96	240	3, 4	$4 \times s_3, 6 \times s_4$	816
4	72	72	4	$8 \times s_4$	576
5	72	72	5, 7	$3 \times s_5, 4 \times s_7$	504
6	144	144	6, 7	$2 \times s_6, 2 \times s_7$ $1 \times s_6, 4 \times s_7$	432
7	72	72	7	$4 \times s_7$	288
8	72	72	8	$3 \times s_8$	216
9	144	144	9	$1 \times s_9$	144

Table 5.7: The dual solution, the patterns that need to be considered for verifying its feasibility, together with their weight and the knapsack capacity.

its feasibility, note that the dual LP constraint

$$-\sum_{j=1}^{k} \mu_j \cdot \lim_{n\to\infty} \frac{\mathrm{OPT}(L1 \dots L_j)}{n} \leq 1$$

is satisfied with equality. It remains to check the other dual constraints, where again we have to test whether $\max_{p\in T_j} w(p) = \max_{p\in T_j} \sum_{i=j}^{k} \lambda_i p_i \leq -\sum_{i=j}^{k} \mu_i$ for all $j = 1, \dots, k$. For solving the associated knapsack problem, we use the following dominance relations to simplify our task:

$$\begin{array}{lll} s_1 \succ s_2 & (2\times 1)s_2 \succ s_3 & (1\times 2)s_1 \succ s_4 \\ s_4 \succ s_5 & (2\times 1)s_5 \succ s_6 & s_4 \succ s_7 \\ s_7 \succ s_8 & (2\times 1)s_8 \succ s_9 & \end{array}$$

We list the optimal patterns to be considered for the knapsack problem in table 5.7. In the following lemmas, we will prove that it suffices to consider these patterns.

Lemma 16. *For T_2, the patterns $p^{(1)} = (0,6,0,8,0,\dots,0)$, $p^{(2)} = (0,12,0,4,0,\dots,0)$, and $p^{(3)} = (0,18,0,\dots,0)$ maximize $w(p) = \sum_{i=2}^{k} \lambda_i p_i$, given the λ_i-values from table 5.7.*

Proof. In this proof, we abbreviate patterns by listing only their second and fourth components. Any vertical line through a bin can intersect with at most two type 4 items, and any horizontal line with at most four. By arranging the items in two rows of four, we see that $(0,8)$ is a dominent pattern. There are only two options for a horizontal line in a bin that contains only type 2 and type 4 items: Either, it crosses (at most) four type 4 items, or it crosses at most three items (of any type). In general, if a bin contains eight type 4 items, there is at least a height of $1/3 - 2\varepsilon$ where horizontal

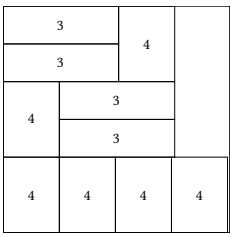


Figure 5.5: A feasible packing for pattern p from lemma 17

lines cross with at most three items (this height can be more if the type 4 items are not exactly aligned). This in turn implies that a volume of at least $(1/3-2\varepsilon)(1/4-300\delta)$ must remain empty in any bin that contains eight type 4 items, as the maximum total width of a set of three items is $3/4+300\delta$. It follows immediately that $(6,8)$ is a dominant pattern, as the free space in the packing tends to exactly $1/3\times 1/4$ if $\varepsilon\to 0$ and $\delta\to 0$ (and since it is indeed a pattern).

If there is a total height of more than $1/3+\varepsilon$ at which a horizontal line intersects with four items, then by considering the highest and the lowest such line, we can identify eight distinct type 4 items. Therefore, in a bin with four to seven type 4 items, at a height of at least $2/3-\varepsilon$, a horizontal line intersects with at most three items, since you can only have one row of four type 4 items. Therefore, in such a bin, there must be $(2/3-\varepsilon)(1/4-300\delta)$ of empty space. We find the following patterns: $(6,7),(8,6),(10,5),(12,4)$ with weights $\frac{792}{413}$, $\frac{816}{413}$, $\frac{840}{413}$, $\frac{864}{413}$, respectively.

If there are at most three type 4 items, then *any* horizontal line intersects with at most three items, the empty space is at least $1/4-300\delta$, and the patterns are $(12,3),(14,2),(16,1),(18,0)$ with weights $\frac{792}{413}$, $\frac{816}{413}$, $\frac{840}{413}$, $\frac{864}{413}$, respectively. □

Lemma 17. *For T_3, the pattern $p=(0,0,4,6,0,\ldots,0)$ maximizes $w(p)=\sum_{i=2}^{k}\lambda_i p_i$, given the λ_i-values from table 5.7.*

Proof. In this proof, we again abbreviate patterns by listing only their third and fourth components. First of all, see fig. 5.5 for a feasible packing of p. Horizontal lines can only intersect with these sets of items: Either three or four type 4 items, or at most *two* items which have total width at most $3/4+170\delta$. As above, $(0,8)$ is a dominant pattern. There is at least a height of $1/3-2\varepsilon$ at which horizontal lines intersect with at most two items. There is at most a height of $1/3-2\varepsilon$ at which horizontal lines intersect with at most one type 4 item.

Since two type 3 items cannot be placed next to each other, this implies that $(2,8)$ is a (dominant) pattern, but with a smaller weight of $\frac{768}{413}$.

If there are seven type 4 items, again there is at most a height of $1/3-2\varepsilon$ at which horizontal lines intersect with at most one type 4 item, so $(3,7)$ is not a pattern. If there are four to six type 4 items, there is an empty volume of at least $(2/3-\varepsilon)(1/4-170\delta)$, so $(4,6)$ is dominant. Moreover, there is at least a height of $2/3-\varepsilon$ at which horizontal lines intersect with at most two items, so $(5,5)$ and $(5,4)$ are not patterns.

If there are three type 4 items, the empty volume is at least $1/4-170\delta$, so $(6,3)$ is a dominant pattern with weight $\frac{792}{413}$. Finally, no bin can contain more than six type 3 items, so no other pattern can be heavier. □

Lemma 18. *For T_5, the pattern $p = (0,0,0,0,3,0,4,0,0)$ maximizes $w(p) = \sum_{i=2}^{k} \lambda_i p_i$, given the λ_i-values from table 5.7.*

Proof. Note that $\lambda_5 = \lambda_7$. Therefore the only question here is how many items of these types can be packed together in a bin. No more than four type 7 items can be packed in any bin, and at most three type 5 items can be packed with them (using similar arguments as above). Moreover, no more than six type 5 items can be packed in any bin, and if there are less than four type 7 items in any bin, then any horizontal line in such a bin intersects with at most three items. Thus at most seven items can be packed into any such bin. □

Lemma 19. *For T_6, the patterns $p^{(1)} = (0,0,0,0,0,2,2,0,0)$ and $p^{(2)} = (0,0,0,0,0,1,4,0,0)$ maximize $w(p) = \sum_{i=2}^{k} \lambda_i p_i$, given the λ_i-values from table 5.7.*

Proof. For a packing of $p^{(1)}$, see fig. 5.4c (we can replace one type 8 item by one type 7 item easily). Observe that a bin can never contain more than two s_6-items, and together with the fact that it is easy to see that no more than two s_7-items can be added to them, it follows that $p^{(1)}$ is a candidate for the heaviest pattern. Likewise, it is clear that no bin can contain more than four s_7-items, and it is easy to see that no more than one s_6-item can be added to those. □

5.2.2 A better lower bound

We believe that our lower bound of 1.859 for rectangle packing could be further improved to 1.907 by extending the input sequence as given in table 5.8. However, we do not have a formal proof of what the heaviest patterns are for the sets T_i (our conjectures are listed in table 5.9).

5.3 Lower bound for Harmonic-type algorithms

Now, we consider the hypercube packing problem in d dimensions, for any $d \geq 2$. We define the class $C^{(h)}$ of Harmonic-type algorithms analogous to [95]. An algorithm $\mathcal{A}$ in $C^{(h)}$ for any $h \geq 1$ distinguishes, possibly among others, the following disjoint subintervals

- $\overline{I}_1 = (1 - y_1, 1]$
- $I_{1,j} = \left(1 - y_{j+1}, 1 - y_j\right]$, for every $j \in \{1, \ldots, h\}$
- $\overline{I}_2 = (y_h, 1/2]$
- $I_{2,j} = \left(y_{h-j}, y_{h-j+1}\right]$, for every $j \in \{1, \ldots, h\}$
- $I_\lambda = (0, \lambda]$

for some parameters y_j and λ, where $1/3 = y_0 < y_1 < \ldots < y_h < y_{h+1} = 1/2$ and $0 < \lambda \leq 1/3$. For convenience, we assume that all y_j are rational.

Algorithm $\mathcal{A}$ has to follow the following rules:

1. For each $j \in \{1, \ldots, h\}$, there is a constant m_j s.t. a $1/m_j$-fraction of the items of side length in $I_{2,j}$ is packed $2^d - 1$ per bin ("red items"), the rest are packed 2^d per bin ("blue items").

type j	width w_j	height h_j	$\frac{\mathrm{OPT}(L_1 \ldots L_j)}{n} \cdot 7224$
		Level 1	
1	$1/4 - 30000\delta$	$1/1807 + \varepsilon$	1
2	$1/4 + 10000\delta$	$1/1807 + \varepsilon$	2
3	$1/2 + 20000\delta$	$1/1807 + \varepsilon$	4
		Level 2	
4	$1/4 - 3000\delta$	$1/43 + \varepsilon$	46
5	$1/4 + 1000\delta$	$1/43 + \varepsilon$	88
6	$1/2 + 2000\delta$	$1/43 + \varepsilon$	172
		Level 3	
7	$1/4 - 300\delta$	$1/7 + \varepsilon$	430
8	$1/4 + 100\delta$	$1/7 + \varepsilon$	688
9	$1/2 + 200\delta$	$1/7 + \varepsilon$	1204
		Level 4	
10	$1/4 - 30\delta$	$1/3 + \varepsilon$	1806
11	$1/4 + 10\delta$	$1/3 + \varepsilon$	2408
12	$1/2 + 20\delta$	$1/3 + \varepsilon$	3612
		Level 5	
13	$1/4 - 3\delta$	$1/2 + \varepsilon$	4515
14	$1/4 + \delta$	$1/2 + \varepsilon$	5418
15	$1/2 + 2\delta$	$1/2 + \varepsilon$	7224

Table 5.8: The input items for a hypothesized lower bound of 1.907.

2. No bin contains an item of side length in $I_{1,i}$ *and* an item of side length in $I_{2,j}$ if $i + j \leq h$.

3. No bin contains an item of side length in $\overline{I}_1$ *and* an item of side length in $I_{2,j}$.

4. No bin contains an item of side length in $I_{1,j}$ *and* an item of side length in $\overline{I}_2$.

5. No bin that contains an item of side length in I_λ contains an item of side length in $I_{1,j}, I_{2,j}, \overline{I}_1$ or $\overline{I}_2$.

We will now define $2h + 1$ input instances for the hypercube packing problem in d dimensions, and for each instance we derive a lower bound on the number of bins any $C^{(h)}$-algorithm must use to pack this input.

Every such input instance consists of three types of items. The input will contain N items of side length u, followed by $(2^d - 1)N$ items of side length v and finally followed by MN items of side length t, where u, v, t and M will be defined for each instance differently. We will then show, for every instance, that one u-item, $2^d - 1$ v-items and M t-items can be packed together in one bin, thus the optimal packing for this input uses at most N bins.

j	types to consider	heaviest patterns p from T_j	$w(p) \cdot 516211/516$
1	1	$7224 \times s_1$	7224
2	2, 4	$5418 \times s_2$	5418
3	3,4	$1806 \times s_3, 43 \times s_4$ $84 \times s_3, 166 \times s_4$	4816
4	4	$168 \times s_4$	4704
5	5, 7	$126 \times s_5$	3528
6	6, 7	$42 \times s_6, 7 \times s_7$ $12 \times s_6, 22 \times s_7$	3136
7	7	$24 \times s_7$	2688
8	8, 10	$18 \times s_8$ $6 \times s_8, 8 \times s_{10}$	2016
9	9, 10	$4 \times s_9, 6 \times s_{10}$	1904
10	10	$8 \times s_{10}$	1344
11	11, 13	$3 \times s_{11}, 4 \times s_{13}$	1176
12	12,13	$2 \times s_{12}, 2 \times s_{13}$ $1 \times s_{12}, 4 \times s_{13}$	1008
13	13	$4 \times s_{13}$	672
14	14	$3 \times s_{14}$	504
15	15	$1 \times s_{15}$	336

Table 5.9: We believe that these are the patterns that need to be considered for verifying the feasibility of the dual solution of the 1.907 lower bound.

5.3.1 Instances $1, \dots, h$

Let $\varepsilon > 0$ be arbitrarily small. For every $j \in \{1, \dots, h\}$, we define the following instance of the problem: Let $u = \frac{1+\varepsilon}{2}, v = (1+\varepsilon)y_{h-j}$ and $t = \frac{(1+\varepsilon)y_{h-j}}{2K}$ for some large integer K such that $t \in I_\lambda$ and $\frac{K}{y_{h-j}} \in \mathbb{N}$. Clearly, $u \in I_{1,h}$ and $v \in I_{2,j}$.

In order to show that one u-item, $2^d - 1$ v-items and M t-items can be packed in one bin, we will define anchor points for each size and then place items at some of these such that no two items are overlapping.

There is only one anchor point for u-items, namely $(0, \dots, 0)$, i.e. the origin of the bin. We place one u item there. For items of side length v, we define anchor points as all points having all coordinates equal to $(1+\varepsilon)/2$ or $(1+\varepsilon)/2 - (1+\varepsilon)y_{h-j}$. This defines 2^d anchor points, but an anchor point can only be used for a v-item if at least one coordinate is $(1+\varepsilon)/2$. Hence, we can pack $2^d - 1$ v-items together with the u-item placed before. For an illustration of this packing in two and three dimensions, see fig. 5.6.

For items of side length t, the anchor points are all points with coordinates equal to $i\frac{(1+\varepsilon)y_{h-j}}{2K}$ for $i = 0, \dots, \frac{2K}{y_{h-j}} - 2$, i.e. we have $(\frac{2K}{y_{h-j}} - 1)^d$ anchor points for these items. These anchor points form a superset of all previous anchor points for

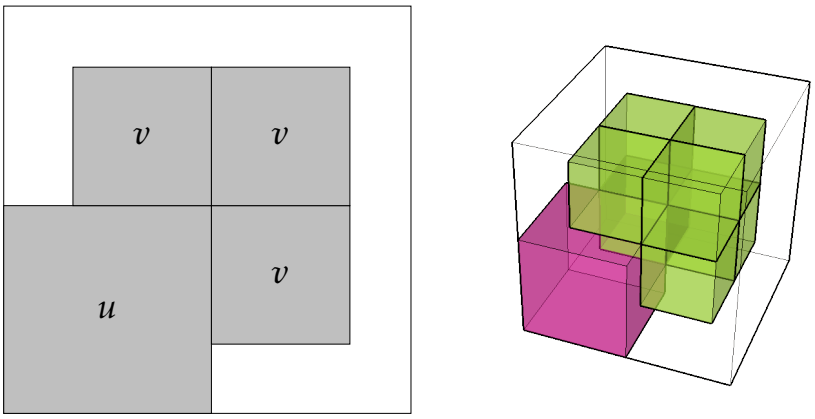


Figure 5.6: Packing of u- and v-items in two and three dimensions for $j = h$, i.e., $v = (1+\varepsilon)/3$.

u- and v-items. Together with the fact that t divides u and v, we can conclude that all larger items take away an integer amount of anchor points for the t-items. To be precise, the u-item blocks $(u/t)^d = \left(K/y_{h-j}\right)^d$ anchor points for t-items and each v-item blocks $\left(\frac{v}{t}\right)^d = (2K)^d$ anchor points for t-items. Hence, we can add $M := \left(\frac{2K-y_{h-j}}{y_{h-j}}\right)^d - \left(\frac{K}{y_{h-j}}\right)^d - (2^d-1)(2K)^d$ t-items to the items packed before.

A Harmonic-type algorithm $\mathcal{A}$ packs a $1/m_j$-fraction of the $N(2^d-1)$ v-items 2^d-1 per bin, using $\frac{(2^d-1)N/m_j}{2^d-1} = \frac{N}{m_j}$ bins in total. The remaining $N(2^d-1)(1-1/m_j)$ v-items are packed 2^d per bin, adding another $N(1-1/m_j)\frac{2^d-1}{2^d} = N(1-1/m_j)\left(1-\frac{1}{2^d}\right)$ bins.

N/m_j of the u-items are added to bins with red v-items, the remaining $N(1-1/m_j)$ items of side length u must be packed one per bin.

Finally, an algorithm in the class $C^{(h)}$ needs at least $NM/\left(\frac{2K-y_{h-j}}{y_{h-j}}\right)^d$ bins to pack the t-items, giving

$$N\left(1-\left(\frac{K}{2K-y_{h-j}}\right)^d-(2^d-1)\left(\frac{2Ky_{h-j}}{2K-y_{h-j}}\right)^d\right)$$

bins for these items. If we let $K\to\infty$, this tends to $N\left(1-1/2^d-(2^d-1)y_{h-j}^d\right)$.

So, the total number of bins needed is at least

$$\begin{aligned}&N\left(\frac{1}{m_j}+\left(1-\frac{1}{m_j}\right)\left(1-\frac{1}{2^d}\right)+1-\frac{1}{m_j}+1-\frac{1}{2^d}-(2^d-1)y_{h-j}^d\right)\\&=N\left(2+\left(1-\frac{1}{m_j}\right)\left(1-\frac{1}{2^d}\right)-\frac{1}{2^d}-(2^d-1)y_{h-j}^d\right)\end{aligned}$$

As the optimal solution uses at most N bins, the competitive ratio of any such algorithm $\mathcal{A}$ must be at least

$$R_{\mathcal{A}} \geq 2+(1-1/m_j)(1-1/2^d)-1/2^d-(2^d-1)y_{h-j}^d \qquad j=1,\dots,h \tag{5.1}$$

5.3.2 Instances $h+1,\dots,2h$

Another set of instances is given for any $j \in \{1,\dots,h\}$, if we use $u = (1+\varepsilon)(1-y_{h-j+1}), v = (1+\varepsilon)y_{h-j}$ and $t = \frac{(1+\varepsilon)y_{h-j}(1-y_{h-j+1})}{K}$ for some large enough integer K such that $u \in I_{1,h-j}, v \in I_{2,j}, t \in I_\lambda$ and $\frac{K}{y_{h-j}}, \frac{K}{1-y_{h-j+1}} \in \mathbb{N}$. For these item sizes, the algorithm is not allowed to combine u-items with v-items in the same bin, although space for items in $I_{1,i}$ with $i > h-j$ is reserved in red bins containing v-items. We define the following anchor points: the point $(0,0)$ for type u; all points with all coordinates equal to $(1+\varepsilon)(1-y_{h-j+1})$ or $(1+\varepsilon)(1-y_{h-j+1}) - (1+\varepsilon)y_{h-j}$ for type v; and all points with all coordinates equal to $i\frac{(1+\varepsilon)y_{h-j}(1-y_{h-j+1})}{K}$ for some $i \in \{0,\dots,\frac{K}{y_{h-j}(1-y_{h-j+1})} - 2\}$ for type t. Again the anchor points for u- and v-items are a subset of the anchor points for t-items, and hence with the same argumentation as before we can pack one u-item together with 2^d-1 v-items and M t-items if we choose $M = \left(\frac{K-y_{h-j}(1-y_{h-j+1})}{y_{h-j}(1-y_{h-j+1})}\right)^d - \left(\frac{K}{y_{h-j}}\right)^d - (2^d-1)\left(\frac{K}{1-y_{h-j+1}}\right)^d$, as the u-item takes up $\left(K/y_{h-j}\right)^d$ anchor points of the t-items and each v-item takes up $\left(K/(1-y_{h-j+1})\right)^d$ of these anchor points.

A similar calculation to before can be done: An algorithm in class $C^{(h)}$ needs $N/m_j + N(1-1/m_j)(1-1/2^d)$ bins for red and blue items of type v. It needs N bins for u-items, as they are packed one per bin, and finally

$$\begin{aligned}
&\frac{NM}{\left(\frac{K-y_{h-j}(1-y_{h-j+1})}{y_{h-j}(1-y_{h-j+1})}\right)^d} \\
&= N\left(1 - \left(\frac{K(1-y_{h-j+1})}{K-y_{h-j}(1-y_{h-j+1})}\right)^d - (2^d-1)\left(\frac{Ky_{h-j}}{K-y_{h-j}(1-y_{h-j+1})}\right)^d\right) \\
&\xrightarrow{K\to\infty} N\left(1 - \left(1-y_{h-j+1}\right)^d - (2^d-1)y_{h-j}^d\right)
\end{aligned}$$

bins are required to pack the t-items. Hence, we need at least

$$\begin{aligned}
&N\left(\frac{1}{m_j} + \left(1-\frac{1}{m_j}\right)\left(1-\frac{1}{2^d}\right) + 1 + 1 - \left(1-y_{h-j+1}\right)^d - (2^d-1)y_{h-j}^d\right) \\
&= N\left(2 + \frac{1}{m_j} + \left(1-\frac{1}{m_j}\right)\left(1-\frac{1}{2^d}\right) - \left(1-y_{h-j+1}\right)^d - (2^d-1)y_{h-j}^d\right)
\end{aligned}$$

bins in total. This gives the following lower bound for the competitive ratio:

$$R_{\mathcal{A}} \geq 2 + 1/m_j + \left(1-1/m_j\right)\left(1-1/2^d\right) - \left(1-y_{h-j+1}\right)^d - (2^d-1)y_{h-j}^d \qquad (5.2)$$
$$j = 1,\dots,h$$

5.3.3 Instance $2h+1$

Let $u = \frac{1+\varepsilon}{2}, v = (1+\varepsilon)y_h$ and $t = \frac{(1+\varepsilon)y_h}{2K}$ for some large enough integer K such that $u \in I_{1,h}, v \in \bar{I}_2, t \in I_\lambda$ and $\frac{K}{y_h} \in \mathbb{N}$. For these item sizes, the algorithm is not allowed to combine u-items with v-items in the same bin. We define anchor points as follows: $(0,0)$ for type u; all points with coordinates equal to $\frac{1+\varepsilon}{2}$ or $\frac{1+\varepsilon}{2} - (1+\varepsilon)y_h$

for type v; all points with coordinates equal to $i\frac{(1+\varepsilon)y_h}{2K}$ for type t. As before, the anchor points for u and v-items are a subset of the t-items' anchor points, and so we can pack one u-item together with 2^d-1 v-items and M t-items if we choose $M = \left(\frac{2K-y_h}{y_h}\right)^d - \left(\frac{K}{y_h}\right)^d - (2^d-1)(2K)^d$.

For this input, any Harmonic-type algorithm uses at least N bins for u-items, $N\frac{2^d-1}{2^d} = N(1-\frac{1}{2^d})$ bins for v-items and $\frac{NM}{\left(\frac{2K-y_h}{y_h}\right)^d}$ bins for t-items. This gives in total

$$\begin{aligned}
&N\left(2-\frac{1}{2^d}+1-\left(\frac{K}{2K-y_h}\right)^d-(2^d-1)\left(\frac{2Ky_h}{2K-y_h}\right)^d\right)\\
&\xrightarrow{K\to\infty} N\left(3-\frac{1}{2^{d-1}}-(2^d-1)y_h^d\right)
\end{aligned}$$

bins. We therefore can derive the following lower bound on the competitive ratio:

$$R_{\mathcal{A}} \geq 3 - 1/2^{d-1} - (2^d-1)y_h^d \tag{5.3}$$

5.3.4 Combined Lower Bound

Given a certain set of parameters (y_j and m_j), the maximum of the three right sides of inequalities (5.1) to (5.3) give us a bound on the competitive ratio of any Harmonic-type algorithm with this set of parameters. In order to get a general (worst-case) lower bound on $R_{\mathcal{A}}$, we need to find the minimum of this maximum over all possible sets of parameters.

This lower bound for $R_{\mathcal{A}}$ is obtained when equality holds in all of inequalities (5.1) to (5.3). To see this, consider the following: We have $2h+1$ variables and $2h+1$ constraints. For $j \in \{1,\dots,h\}$, we see that ineq. (5.1) is increasing in m_j and ineq. (5.2) is decreasing in m_j. Next, let $c \in \{1,\dots,h-1\}$. We see that ineq. (5.1) for $j = h-c \in \{1,\dots,h-1\}$ is decreasing in y_c, and ineq. (5.2) for $j = h-c+1 \in \{2,\dots,h\}$ is increasing in y_c. Finally, we have that ineq. (5.2) for $j=1$ is increasing in y_h and ineq. (5.3) is decreasing in y_h. This means, given certain parameters y_j and m_j, if e.g. ineq. (5.3) gives a smaller lower bound on $R_{\mathcal{A}}$ than ineq. (5.2) with $j=1$ does, we can decrease the value of y_h such that the maximum of the three lower bounds becomes smaller.

Setting the right hand side of ineq. (5.1) equal to the right hand side of ineq. (5.2), gives us $\frac{1}{m_j} = (1-y_{h-j+1})^d - \frac{1}{2^d}$ or alternatively $\frac{1}{m_{h-j+1}} = (1-y_j)^d - \frac{1}{2^d}$. Plugging this into ineq. (5.1) (replacing j by $h-j+1$), we find that

$$y_j = 1 - \left(\frac{-2^d R_{\mathcal{A}} + 2^d y_{j-1}^d - 4^d y_{j-1}^d - 1 + 3\cdot 2^d - 1/2^d}{2^d-1}\right)^{1/d} \tag{5.4}$$

Recall that we require $1/3 = y_0 < y_1$. From this, combined with eq. (5.4) for $j=1$, we obtain that

$$R_{\mathcal{A}} \geq 3 - 2\frac{2^d-1}{3^d} - \frac{2^d+1}{4^d}$$

We list some values of the lower bound for several values of d in table 5.10.

Note that for $d=1$, our formula yields the bound of Ramanan et al. [95]. Surprisingly, it does not seem to help to analyze the values of $y_2,\dots,y_h$. Especially, equations

d	1	2	3	4	5	6	∞
$R_{\mathcal{A}} >$	1.58333	2.02083	2.34085	2.56322	2.71262	2.81129	3

Table 5.10: Lower bounds for Harmonic-type algorithms in dimensions 1 to 6 and limit for $d \to \infty$.

involving y_j for $j > 1$ become quite messy due to the recursive nature of eq. (5.4). If h is a very small constant like 1 or 2, we can derive better lower bounds for $R_{\mathcal{A}}$. For larger h, we can use the inequalities $y_1 < y_h, y_2 < y_h, y_3 < y_h$ (i.e. assuming that $h > 3$) to derive *upper* bounds on the best value $R_{\mathcal{A}}$ that could possibly be proven using this technique. These upper bounds are very close to 2.02 and suggest that for larger h, an algorithm in the class $C^{(h)}$ could come very close to achieving a ratio of 2.02 for these inputs. However, since the inequalities become very unwieldy, we do not prove this formally.

Theorem 5. *No Harmonic-type algorithm for two-dimensional online hypercube packing can achieve an asymptotic competitive ratio better than* 2.0208.

5.4 Further lower bounds

Inspired by Extreme Harmonic as described in chapter 4, one could try to improve online algorithms for packing 2-dimensional squares by incorporating two ideas from the one-dimensional case: combining large items (i.e. items larger than 1/2) and medium items (i.e. items with size in $(1/3, 1/2]$) whenever they fit together (ignoring their type), and postponing the coloring decision. The former is intuitive, while the idea of the latter would be the following: When items of a certain type arrive, we first give them provisional colors and pack them into separate bins (i.e. one item per bin). After several items of this type arrived, we choose the smallest of them to be red and all others are colored blue. With following items of this type, we fill up the bins with additional items. However, simply adding two more red items to the bin with a single red item might be problematic: When filling up the red bins with two more red items, it could happen that these later red items are larger than the first one - negating the advantage of having the first red item be relatively small. Alternatively, we could leave the red item alone in its bin. This way, we make sure that at most 3/4 of the blue items of a certain medium type are smaller than the smallest red item of this type, but we have more wasted space in this bin.

For both approaches discussed above we will show lower bounds on the competitive ratio that are only slightly lower or even higher than the lower bound established in section 5.3 for Harmonic-type algorithms.

5.4.1 Always combining large and medium items

First, we consider algorithms that combine small and large items whenever they fit together. We define a class of algorithms B_1 that distinguish, possibly among others, the following disjoint subintervals (types):

- $I_m = (1/3, y]$ for some $y \in (1/3, 1/2]$
- $I_\lambda = (0, \lambda]$

These algorithms satisfy the following rules:

1. There is a parameter α s.t. an α-fraction of the items of side length in I_m are packed 3 per bin ("red items"), the rest are packed 4 per bin ("blue items").
2. No bin that contains an item of side length in I_λ contains an item of side length larger than $1/2$ or an item of side length in I_m.
3. Items of type I_m are packed without regard to their size.

Let $a, b \in I_m, a < b$. We consider two different inputs, both starting with the same set of items: $\frac{\alpha}{3}N$ items of size b and $(1-\alpha/3)N$ items of size a (i.e. in total N items of size a and b). By rule 3, the adversary knows beforehand which item will be packed in which bin, as they belong to the same type. Hence, the adversary can order these items in such a way that the items colored blue by the algorithm are all a-items, and in each bin with red items, there are two a- and one b-item. By rule 1, the online algorithm uses $(\frac{\alpha}{3} + \frac{1-\alpha}{4})N = \frac{3+\alpha}{12}N$ bins for items of this type.

The sizes a and b will tend towards $1/3$, as this way the adversary can maximize the total volume of sand (infinitesimally small items) that can be added to any bin in the optimal solution while not changing the way the algorithm packs these items and increasing the number of bins the algorithm needs for packing the sand items. Therefore, we will assume that a and b are arbitrarily close to $1/3$.

In the first input, after these medium items, $\frac{(1-\alpha/3)N}{3}$ items of size $1-a$ arrive, followed by sand of total volume $\frac{24+7\alpha}{324}N$. In the optimal solution, we can pack $\frac{\alpha}{12}N$ bins with four b-items and sand of volume $5/9$ each, and $\frac{(1-\alpha/3)N}{3}$ bins with three a-items, one $(1-a)$-item and sand of volume $\frac{\alpha}{12}N \cdot \frac{2}{9}$ each. Hence, the optimal solution uses $\frac{\alpha}{12}N + \frac{(1-\alpha/3)N}{3} = \frac{12-\alpha}{36}N$ bins.

The algorithm, however, cannot pack a large item into any of the bins with red medium items, as these always contain a b-item. Hence, in addition to the $\frac{3+\alpha}{12}N$ bins for medium items, the algorithm needs $\frac{(1-\alpha/3)N}{3}$ bins for large items and at least $\frac{24+7\alpha}{324}N$ bins for sand. This gives in total at least $\frac{213-2\alpha}{324}N$ bins, and a competitive ratio of at least

$$\frac{\frac{213-2\alpha}{324}N}{\frac{12-\alpha}{36}N} = \frac{213-2\alpha}{9(12-\alpha)} \tag{5.5}$$

In the second input, after the medium items, $N/3$ items of size $1/2+\varepsilon$ will arrive, followed by sand of total volume $\frac{5}{36}N$. The optimal solution packs all medium items three per bin, using $N/3$ bins, and adds one large item and sand of volume $15/36$ in each such bin. In the algorithm's solution, large items can only be added to the $\alpha N/3$ bins containing three red items, i.e. it needs additional $N/3 - \alpha N/3$ bins for the remaining $N/3 - \alpha N/3$ large items. Finally, the algorithm uses at least $5/36N$ bins for sand. The algorithm therefore uses in total at least $\frac{3+\alpha}{12}N + (1-\alpha)N/3 + 5/36N = \frac{26-9\alpha}{36}N$ bins. This gives a competitive ratio of at least

$$\frac{3(26-9\alpha)}{36} = \frac{26-9\alpha}{12} \tag{5.6}$$

Observe that eq. (5.5) is increasing in α, while eq. (5.6) is decreasing in α. Hence, the minimum over the maximum of the two bounds is obtained for the α-value that makes both bounds equal, which is $\alpha = \frac{197-\sqrt{36541}}{27} \approx 0.2164$. For this α, both bounds become larger than 2.0043.

Theorem 6. *No algorithm in class B_1 for two-dimensional online hypercube packing can achieve a competitive ratio of less than* 2.0043.

5.4.2 Packing red medium items one per bin, postponing the coloring

Now, consider the algorithm that packs red items alone into bins and makes sure that at most 3/4 of the blue items of a certain type are smaller than the smallest red item of this type. We define a new class of algorithms B_2 that distinguishes, possibly among others, the following disjoint subintervals (types):

- $I_m = (1/3, y]$
- $I_\lambda = (0, \lambda]$

Furthermore, algorithms in B_2 satisfy the following rules:

1. There is a parameter α s.t. an α-fraction of the items of side length in I_m are packed 1 per bin ("red items"), the rest are packed 4 per bin ("blue items").
2. No bin that contains an item of side length in I_λ contains an item of side length larger than 1/2 or an item of side length in I_m.
3. Items of side length in I_m are initially packed one per bin. At some regular intervals, the algorithm fixes some of these items to be red, and does not pack additional items of the same type with them.

From rule 3 we can conclude that the algorithm gives the following guarantee: 1/4 of the blue items with size in I_m are not smaller than the smallest red item with size in I_m.

Let $a, b \in I_m, a < b$ as before. We again consider two different inputs, both starting with the same set of items: $\alpha N + \frac{1-\alpha}{4}N$ items of size b, and $\frac{3(1-\alpha)}{4}N$ items of size a. They arrive in such an order that all red items are b-items, and all bins with blue items contain one b- and three a-items. We require the b-item in the blue bins because of the postponement of the coloring: If the first blue item in a bin was an a-item, the algorithm would choose this item to become red and not one of the b-items. By rule 1, the algorithm needs $\frac{1-\alpha}{4}N + \alpha N = \frac{1+3\alpha}{4}N$ bins for these N items.

In the first input, after the medium items arrived, we get $\frac{1-\alpha}{4}N$ large items of size $1-a$, followed by sand of total volume $\frac{13+7\alpha}{144}N$. The optimal solution can pack the a-items three per bin together with one $(1-a)$-item, using $\frac{1-\alpha}{4}N$ bins for these items. The b-items are packed four per bin, using $(\frac{\alpha}{4} + \frac{1-\alpha}{16})N$ bins. Note that the empty volume in all bins of these two types is $\frac{1-\alpha}{4}N \cdot \frac{2}{9} + (\frac{\alpha}{4} + \frac{1-\alpha}{16})N \cdot \frac{5}{9} = \frac{13+7\alpha}{144}N$, i.e. it equals exactly the volume of the sand, so the sand can be filled in these holes without using further bins. Hence, the optimal number of bins is $\frac{1-\alpha}{4}N + (\frac{\alpha}{4} + \frac{1-\alpha}{16})N = \frac{5-\alpha}{16}N$.

The algorithm uses, as discussed before, $\frac{1+3\alpha}{4}N$ bins for the medium items of size a and b. The large items cannot be added to red medium items, as they do not fit together, thus the algorithm uses $\frac{1-\alpha}{4}N$ additional bins for the large items. Finally, according to rule 2, at least $\frac{13+7\alpha}{144}N$ additional bins are needed to pack the sand. This gives in total at least $\frac{1+3\alpha}{4}N + \frac{1-\alpha}{4}N + \frac{13+7\alpha}{144}N = \frac{85+79\alpha}{144}N$ bins. We find that the competitive ratio is at least

$$\frac{\frac{85+79\alpha}{144}N}{\frac{5-\alpha}{16}N} = \frac{85+79\alpha}{9(5-\alpha)} \tag{5.7}$$

In the second input, $N/3$ items of size $1/2 + \varepsilon$ arrive after the medium items, followed by sand of total volume $5/36N$. The algorithm packs this input the same way as a B_1 algorithm, so the analysis carries over. We get a competitive ratio of at least

$$\frac{\frac{26-9\alpha}{36}N}{N/3} = \frac{26-9\alpha}{12} \tag{5.8}$$

It can be seen that eq. (5.7) is a function increasing in α, while eq. (5.8) is decreasing in α, hence the minimum over the maximum of two bounds is reached when they are equal. In that case, $\alpha = \frac{529-\sqrt{274441}}{54} \approx 0.0950$, and the lower bound for the competitive ratio becomes larger than 2.0954.

Theorem 7. *No algorithm in class B_2 for two-dimensional online hypercube packing can achieve a competitive ratio of less than* 2.0954*.*

Note here that this is an even higher lower bound than the one shown in the previous section 5.4.1, although we use postponement of the coloring here. This indicates that the space we waste by packing red medium items separately outweighs the advantage we get by having a guarantee about the size of the red item.

5.5 Discussion and directions for future work

We gave new lower bounds for general algorithms and several specific classes of algorithms for the online bin packing problem in higher dimensions. We also showed that by transfering some of the ideas from the one-dimensional case to the two-dimensional case, one cannot easily gain improved algorithms as similarly high lower bounds can be found.

Lower bounds generally serve two purposes: Providing some guidance which results are worth tackling, and more importantly, giving an intuition on which difficulties prevent algorithms from performing better. Our result on general algorithms mainly serves the first purpose, while especially lower bound results for specific *classes of algorithms* such as the ones presented in sections 5.3 and 5.4 can illustrate the limitations of certain algorithms and inspire new approaches.

To that end, we gave a lower bound for Harmonic-type algorithms as Ramanan et al. [95] defined them. In the second step, we tackled the question of whether our improvements from Extreme Harmonic (see chapter 4), which helped to overcome this exact lower bound in the one-dimensional case, could also help to overcome these lower bounds for higher dimensions. The answer we found is that such Extreme Harmonic-inspired algorithms do indeed *not* seem to allow for improvements even in only two dimensions.

Why is this the case? Our lower bound for Extreme Harmonic-inspired two-dimensional algorithms basically deals with the interesting question of how to avoid that some of the red medium items in a bin are too large to allow large items to be added to this bin. In the one-dimensional case, it was feasible to pack items separately first and then decide on the smallest one to become red. This gave us guarantees about the size of half of the blue items.

In two dimensions, this is not so easy; we basically have *two* possibilities. Either we pack three red medium items into a bin; in that case, it is hard to control the size of the largest of them. Or, alternatively, we pack only one medium red item into a bin. This would give us guarantees of the desired form, however, these guarantees are

much weaker than before, as now we can only make statements about one quarter of the blue items instead of half of them. Additionally, with this approach we also waste much more space in such bins.

In essence, the increased complexity due to the additional dimension means that the additional structural information we gain by postponing the coloring does not suffice to improve our results significantly. For the second suggested approach, we even obtained a higher lower bound than for Harmonic-type algorithms. Thus, here the loss due to wasted space seems to be more severe than the gain we get from item size guarantees.

A possible solution to this problem and therefore an interesting direction for future work could be to fill the space that is wasted when packing only one medium red item with a large item with items of a third, smaller type. However, this would probably require a larger number of weighting functions – possibly also more complex – in order to correctly count all bins. This would finally also increase the complexity of the linear programs that need to be solved. Thus, novel ideas are needed to incorporate such ideas.

Part II

Geometric Knapsack Problems

6 An Overview of Geometric Knapsack

The knapsack problem can be considered a natural dual of the bin packing problem: Instead of packing *all* items into *as few as possible* containers, we now want to pack a *maximum weight subset* of items into a *single* container. In order to illustrate this connection, let us again describe the cutting stock problem from chapter 3. A merchant offers some sheet material such as sheet metal and has a supply of sheets of fixed dimensions. Customers request rectangular cutouts of certain dimensions which are at most the dimensions of one sheet. While one can see this as a bin packing problem as described in chapter 3, one can also imagine knapsack-type questions, e.g., what is the maximum profit one can make from a single sheet of material. This also illustrates that the knapsack problem is often a subproblem (e.g. in form of a separation problem for linear programs) when considering bin packing problems; note that we also used one-dimensional knapsack problems for our result in chapter 4.

The one-dimensional knapsack problem has a long history because of its fundamental question: Among a set of alternatives, choose (binarily) the maximum profit subset, while adhering to some resource constraints. One can imagine a huge number of practical examples for this problem: For example, consider a financial decision problem, where given a collection of investment opportunities, each with a (expected) profit and cost (the sum you need to invest), choose a subset with maximum expected profit subject to a budget constraint. There are other applications in e.g. cargo loading or cryptography; see Kellerer et al. [81] for more detailed examples. The number of example applications one can imagine for the geometric two-dimensional problem is of course similarly vast. Recall for example the advertisement placement problem from the introduction; for literature on this and similar problems, see e.g. Freund and Naor [47].

6.1 Problem definition

The one-dimensional knapsack problem asks the following question: Given a knapsack of unit size and a set of items I, each item i with a particular size s_i and a profit or weight p_i, select a maximum profit subset $I' \subseteq I$ of these items that fit into the knapsack, that is, $\sum_{i \in I'} s_i \leq 1$.

As for bin packing, we might consider two different generalizations of this problem to higher dimensions. In the *two-dimensional (vector) knapsack problem*, each item i has d sizes $s_i^1, \ldots, s_i^d$ for some dimension d and we have d capacities $c_1, \ldots, c_d$. Our objective is to select a subset of items I' such that $\sum_{i \in I'} s_i^j \leq c_j$ for any $j \leq d$. This setting represents the generalization where we have a number of restricted resources (e.g. size, weight) and each item requests a certain amount of these resources.

In the two-dimensional *geometric* knapsack problem, the knapsack is – in the most general setting – a rectangle of size $N \times N'$, and each item i has a particular

width $w_i \leq N$, height $h_i \leq N'$ and profit p_i. The goal is to select a subset of the input items and pack them into the knapsack such that the items are axis-aligned and no two items overlap. In the most general version, items might not be rotated but must be packed with the orientation given in the input. The objective function is to maximize the total profit of the selected items. In this problem, the fullfilment of the constraints cannot be checked independently as in the non-geometric setting; instead, we do not only need to select the items but also specify a position where to pack them. This thesis focuses on the geometric knapsack problem.

For this problem, many variants can and have been considered. We list those relevant for this thesis below:

Square knapsack This is one of the most common problem variants, in which we assume $N = N'$.

Square items All input items are squares. We denote by s_i the side length of item i.

Rotatable items Items can be rotated by 90 degrees.

Resource augmentation While the optimal solution uses a knapsack of size $N \times N'$, the algorithm is allowed to use a knapsack of size $(1+\varepsilon)N \times (1+\varepsilon)N'$ (two-dimensional resource augmentation) or $(1+\varepsilon)N \times N'$ or $N \times (1+\varepsilon)N'$ (one-dimensional resource augmentation). Here, ε is some small positive constant, and usually the approximation factor and running time of such an algorithm depends on ε.

Unweighted items Items have uniform profits, i.e., all items have profit $p_i = 1$.

In this part we consider the *absolute approximation ratio*, unless stated otherwise.

6.2 Related work

Early literature on the one-dimensional knapsack problem dates back to the 19th century [92], where it was basically shown that a general integer program can be reduced to the knapsack problem, namely by aggregating the constraints into a single knapsack constraint. This illustrates nicely the fundamental importance of the knapsack problem. In-depth research on it started in the 1950s [34]. The problem is NP-complete and there is a vast amount of literature on approximation algorithms for it, peaking in an FPTAS [69]. See Kellerer et al. [81] for an extensive discussion of this problem and many variants of it.

For the two-dimensional vector knapsack problem, a PTAS has been known since the 1980s due to Frieze and Clarke [48]. Later, a PTAS with improved running time was given by Caprara et al. [24]. On the other hand, it has been known since the 1980s that no FPTAS can exist [90], and it is also impossible to obtain an EPTAS [85].

We now discuss in more detail the generalization of this problem to the two-dimensional *geometric* setting. The problem of finding a maximum profit packing of rectangles into a knapsack is NP-hard [87]. Caprara and Monaci [23] were the first to give an approximation algorithm for this problem, the approximation ratio being $3+\varepsilon$. Jansen and Zhang [74] gave a $(2+\varepsilon)$-approximation algorithm for the case that the profits are all equal (also called cardinality case). The same authors gave a $2+\varepsilon$ approximation for the general geometric knapsack problem [75]. Very recently, Gálvez et al. [51] improved this to a $17/9+\varepsilon$ approximation. They also gave

a $\frac{558}{325} + \varepsilon \approx 1.72 + \varepsilon$ approximation for the cardinality case, a $3/2 + \varepsilon$ approximation for the weighted case with rotation and a $4/3 + \varepsilon$ approximation for the cardinality case with rotation. Moreover, there is a PTAS with quasi-polynomial running time (also called QPTAS) when the input is quasi-polynomially bounded by Adamaszek and Wiese [1].

In the special case where all input items are squares, Baker et al. [6] gave a 4/3 approximation for the cardinality case. This was improved to an AFPTAS by Jansen and Zhang [74]. For the weighted case, Harren [62] gave a $5/4 + \varepsilon$ approximation, and a PTAS is known due to Jansen and Solis-Oba [72]. Harren [62] also showed that his algorithm can be generalized to higher dimensions, yielding an approximation ratio of $1 + 1/2^d + \varepsilon$ for d dimensions.

While for the general problem with rectangles, no PTAS is known so far, we can achieve an approximation guarantee of $1 + \varepsilon$ when we allow the algorithm slightly more space in the knapsack than the optimal solution is allowed to use. To be more precise, [46] gave an algorithm that finds a packing of profit $(1 - \varepsilon)$OPT into a knapsack of size $(1 + \varepsilon) \times (1 + \varepsilon)$, where OPT is the optimal profit that can be packed into a 1×1 knapsack. Similarly, in [71], it was shown how to obtain a $1 + \varepsilon$ approximation in a $1 \times (1 + \varepsilon)$ knapsack. We refer to these settings as two-dimensional and one-dimensional resource augmentation, respectively.

When the profit of the items equals their area, Fishkin et al. [46] achieved a PTAS for packing square items and Bansal et al. [16] achieved a PTAS for packing general rectangles.

6.3 Results in this thesis

We present two results regarding geometric knapsack: In chapter 7, we improve the running time of the PTAS for square packing by Jansen and Solis-Oba [72] from $\Omega\left(n^{2^{2^{1/\varepsilon}}}\right)$ to $O_\varepsilon(1)n^{O(1)}$, i.e., we present an EPTAS. In chapter 8, we show how to even obtain an algorithm that returns an *optimal* solution for packing rectangles using two-dimensional resource augmentation, while still using only EPTAS running time – previously, a PTAS was the best result known.

7 Geometric Knapsack with Squares

In this chapter, we discuss the two-dimensional geometric knapsack problem with a square knapsack and square items. Items have different profits and are not rotatable. For this problem, the best known algorithm so far was a PTAS given by Jansen and Solis-Oba [72]. It has a running time of the form $\Omega\left(n^{2^{2^{1/\varepsilon}}}\right)$, i.e., the exponent of n is triple exponential in $1/\varepsilon$. In this chapter, we give an EPTAS for this problem.

For many geometric problems in combinatorial optimization, a common and widely applied technique is the so-called shifting technique. In knapsack problems, for instance, items are classified into large and small items with the intention – simply speaking – that large items are much larger than small items. This classification is done such that the "medium" items inbetween can be neglected since, e.g., they give very little profit in the optimal solution. Producing this gap between the size of large and small items is then used to argue differently about these subsets: Large items can often be enumerated (as there cannot be too many of them) and small items are small enough to simplify packing them (having small items compared to the knapsack size is a much easier setting; see e.g. [45]). However, when using this approach the relative size of the large squares can then be very small, in the order of $\varepsilon^{1/\varepsilon}$, and since most algorithms enumerate them, this leads to a running time whose exponent of n is exponential in $1/\varepsilon$.

This raises the question whether this highly impractical running time is truly necessary or whether better running times are possible, for instance of the form $n^{O(1/\varepsilon)}$ or even $O_\varepsilon(1) \cdot n^{O(1)}$. We answer this question for the geometric knapsack problem with squares by providing an EPTAS. As this problem cannot have an FPTAS [87], we thus achieve the best possible running time up to constant factors.

Results in this chapter and organization Our result largely builds on the previous PTAS by Jansen and Solis-Oba [72]. We thus briefly describe the main ideas behind their algorithm in section 7.1. In the subsequent sections 7.2 to 7.4, we then show how to improve the running time of some steps of this PTAS in order to obtain our EPTAS. In the final section 7.5, we present our EPTAS as a whole and summarize the total running time.

7.1 PTAS by Jansen and Solis-Oba

The PTAS by Jansen and Solis-Oba [72] follows a general approach that many algorithms for geometric packing problems implement. First, they give a structural result, showing that a near-optimal packing P^* with a specific structure must exist (near-optimal means having approximation ratio $1+\varepsilon$). Then, they argue how one can enumerate all such structured packings in polynomial time, implying that one can find a solution which is at least as good as P^* and thus a $(1+\varepsilon)$-approximation.

Our EPTAS builds on their approach, and we show how some of their crucial steps can be speeded up significantly by examining the input items more carefully.

7.1.1 Prerequisites

Before diving into the result by Jansen and Solis-Oba [72], we will first clarify some terminology and techniques used in this chapter, which are very common in the community of geometric packing problems, but might not be obvious to readers not familiar with such results. For showing the structural result – i.e., that there exists a near-optimal packing with a certain structure – one usually starts from some optimal packing and then changes the packing to obtain the desired structure. Such changes might e.g. include removal of items, where we need to make sure that the profit of the removed items is at most a $\frac{1}{1+\varepsilon}$-fraction of the total profit of the packing. We might also change the input by e.g. rounding down the weights of some input items. This is possible as long as the values to which we want to round them can be computed in polynomial time from the input.

A second terminology often used is that of *guessing a certain parameter*. This simply means that we enumerate all possibile choices for this parameter and continue our algorithm with these choices; i.e., we branch our algorithm's execution. We then know that in at least one of these branches we "guessed" the right value, and thus, if we return in the end the best result of all branches, we are at least as good as if we knew the right choice of this parameter right from the start and have used this value.

The last prerequisite we want to mention here is the very basic knapsack algorithm *Next-Fit Decreasing Height* (NFDH). Items are packed from left to right into rows or shelves (rectangular areas with the same width as the knapsack) and the shelves are stacked on top of each other within the knapsack. Within a shelf, the items are aligned with the bottom of the shelf. There is always a current shelf, initially its bottom line is the bottom of the knapsack. The algorithm sorts the items in decreasing order of height (for squares, this is of course the same as sorting by width, but this algorithm can also be used for packing rectangles) and then processes each item in this order. If the current item can be packed into the current shelf (as far left as possible, next to the last packed item), pack it there. Otherwise, open a new shelf with bottom line equal to the top of the first item in the current shelf, make this new shelf the current shelf, and pack the item there. Stop packing when the next item cannot be packed anymore into the current shelf and there is not enough space for opening a new shelf. A packing resulting from NFDH is illustrated in fig. 7.1. We will later use the following lemma about NFDH which follows directly from lemma 5.1 in Jansen and Solis-Oba [72]; intuitively, NFDH waists almost no area in the knapsack if the items are small compared to the knapsack size:

Lemma 20. *Let S be a set of squares each of size at most δ and let R be a rectangular area of length a and width b. NFDH either packs in R all squares from S, or it packs squares of total area at least $(b - 2\delta) \times (a - \delta)$.*

7.1.2 KR-packings

Jansen and Solis-Oba [72] show the existence of a near-optimal packing using the algorithm due to Kenyon and Rémila [82]. We refer to the type of packing resulting from Kenyon and Rémila's algorithm as a KR-packing, see fig. 7.2 for a sketch. Intuitively, when packing squares into a rectangular region that has much larger

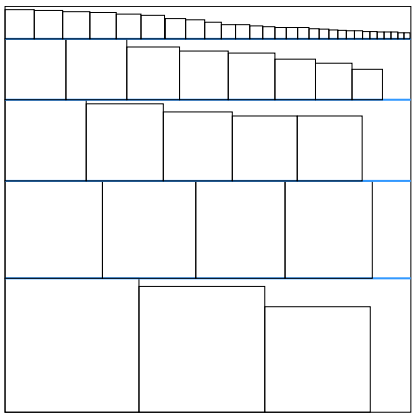


Figure 7.1: A packing produced by NFDH. The blue lines show the bottom lines of the shelves.

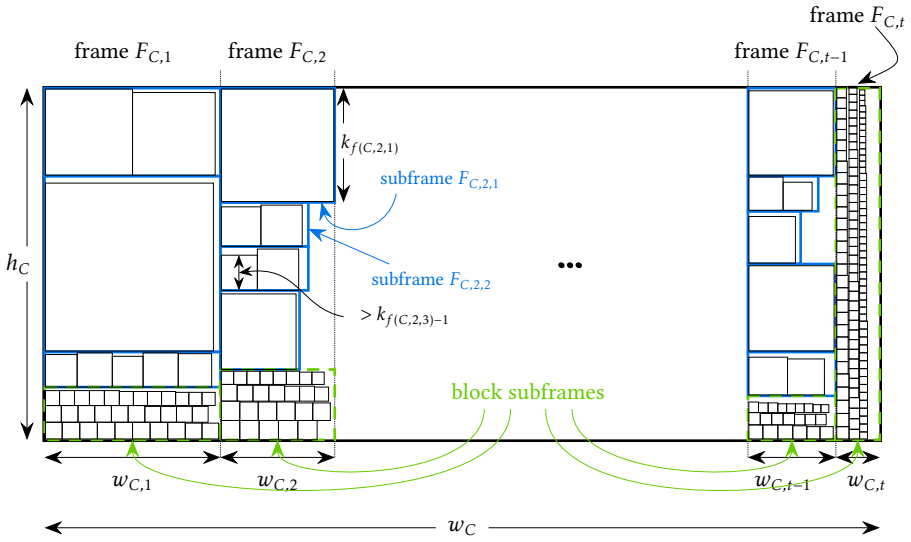


Figure 7.2: Illustration of a KR-packing with t frames.

width than height[1], the KR-algorithm partitions this region into frames (vertical slots within the region), and each frame into subframes (horizontal slots within the frame). There are two types of such subframes: row subframes, which contain squares of roughly the same size; and block subframes, which contain squares that are very small compared to the size of the subframe. In addition, for the heights of the row subframes, there are only few possible values, and within each row subframe, squares are only packed in one row from left to right (i.e. any vertical line going through a row subframe intersects at most one square).

[1]Kenyon and R'emila's algorithm was originally intended for solving the strip packing problem.

Definition 24 (KR-packing)**.** *Consider a rectangular area C with height h_C and width w_C and a set of squares S. Assume w.l.o.g. that $h_C \leq w_C$. We say that the squares in S are packed into C in a* KR-packing *if*

- *there are values $k_0, \ldots, k_{1/\varepsilon^4} \in \mathbb{R}$ with $k_0 = \varepsilon^4 \cdot h_C \leq k_1 \leq \ldots \leq k_{1/\varepsilon^4}$ such that each k_i is the size of some input item,*
- *the area C is partitioned (vertically) into at most $(\frac{2+\varepsilon^2}{\varepsilon^2})^2 = O(1/\varepsilon^4)$ rectangular* frames *$F_{C,1}, F_{C,2}, \ldots$ of height h_C and widths $w_{C,1}, w_{C,2}, \ldots$,*
- *each frame $F_{C,\ell}$ is (horizontally) partitioned into at most $1/\varepsilon^2$ subframes $F_{C,\ell,1}, F_{C,\ell,2}, \ldots$. For each such subframe $F_{C,\ell,j}$ denote by $w_{C,\ell,j}$ its width and by $h_{C,\ell,j}$ its height. We have that $w_{C,\ell,j} \geq \varepsilon^2 \cdot h_C$. For a subframe $F_{C,\ell,j}$ we have that either*
  1. *there is an index $f(C,\ell,j) \in \{1, \ldots, 1/\varepsilon^4\}$ such that $h_{C,\ell,j} = k_{f(C,\ell,j)}$, each vertical line crossing $F_{C,\ell,j}$ intersects at most one square i packed in $F_{C,\ell,j}$, and for this square i we have that $s_i \in (k_{f(C,\ell,j)-1}, k_{f(C,\ell,j)}]$ and if $F_{C,\ell,j}$ contains $\overline{w}_{C,\ell,j}$ such squares then $w_{C,\ell,j} \geq \overline{w}_{C,\ell,j} \cdot k_{f(C,\ell,j)}$ (a* row *subframe or a* row *for short) or*
  2. *it contains only squares i with $s_i \leq \varepsilon^4 \cdot h_C$ and $h_{C,\ell,j} \geq \varepsilon^2 \cdot h_C$ (a* block *subframe or a* block *for short) or*
  3. *it does not contain any squares (an* empty *subframe).*
- *each frame $F_{C,\ell}$ has at most one subframe which is a block or empty.*

We define KR-packings for areas C with $h_C > w_C$ (*vertical KR-packings*) in a similar manner. For the sake of simplicity of presentation, we will only speak of horizontal KR-packings (as defined in definition 24) in this chapter, but every statement about them also holds analogously for vertical KR-packings.

Note that all squares i with $s_i > \varepsilon^4 \cdot h_C$ are packed into row subframes and all squares i' with $s_{i'} \leq \varepsilon^4 \cdot h_C$ are packed into block subframes; that is, given a rectangular area that is packed with a KR-packing and a set of items that shall be packed into it, we can immediately partition the input items into two sets which are packed in different subcontainers of the packing.

7.1.3 Structured packing

We can now formulate the structural result by Jansen and Solis-Oba:

Lemma 21 (Structured packing [72])**.** *For any instance there exists a solution consisting of a set of squares $I' \subseteq I$ such that $p(I') \geq (1 - O(\varepsilon))\mathrm{OPT}$ and there is a packing for I' with the following properties:*

- *there is a subset $I'_L \subseteq I'$ of squares with $|I'_L| \leq F(\varepsilon)$ that are labeled as* big squares*, where $F(\varepsilon) = (1/\varepsilon)^{2^{O(1/\varepsilon^4)}}$,*
- *the space not occupied by the big squares can be partitioned into at most $F(\varepsilon)$ rectangular cells $\mathcal{C}$,*
- *each square in $I'_S := I' \setminus I'_L$ is packed into a cell such that it does not overlap any cell boundary,*

- *each cell in $\mathcal{C}$ is either labeled as a* block *cell or as an* elongated *cell. Denote its height by h_C and its width by w_C.*
  - *For each block cell $C \in \mathcal{C}$ we have that each square $i \in I'_S$ packed into C satisfies $s_i \leq \varepsilon^2 \cdot \min\{h_C, w_C\}$.*
  - *For each elongated cell $C \in \mathcal{C}$ we have that the squares in it are packed in a KR packing.*

Unfortunately, $F(\varepsilon)$ can be as large as $(1/\varepsilon)^{2^{O(1/\varepsilon^4)}}$. Using lemma 21 one can obtain a PTAS with a running time of $n^{O(F(\varepsilon))} = n^{O_\varepsilon(1)}$: first, we guess the at most $F(\varepsilon)$ large squares in time $n^{O(F(\varepsilon))}$. Secondly, we guess the sizes of the cells in $\mathcal{C}$. This can be done directly in pseudopolynomial running time $(n \cdot \max_i s_i)^{O(F(\varepsilon))}$ or in time $n^{O(F(\varepsilon))}$ by exploiting some further properties of the decomposition (not stated above, see [72] for details). Then we compute a packing for the guessed large squares and the cells. It remains to select some of the remaining squares and to pack them into the cells. To this end, we guess the heights and widths of all frames and subframes in each elongated cell (again, either directly in pseudo-polynomial time or in polynomial time using some further properties of the decomposition). This splits each elongated cell into a set of rows and a set of blocks. It remains to pack squares such that

- into each block cell or block subframe C we pack only squares $i \in I$ with $s_i \leq \varepsilon^2 \cdot \min\{h_C, w_C\}$ such that the total area of the assigned squares does not exceed the area of C
- into each row subframe C we pack only squares i with $s_i \leq h_C$. Also, we ensure that if C is contained in an elongated cell C' with $h_{C'} \leq w_{C'}$ then each vertical line intersects at most one square packed into C. (If $h_{C'} \geq w_{C'}$, the same holds for horizontal lines.)

This yields an instance of the generalized assignment problem (GAP) [100] with a constant number of containers (or machines as stated in the terminology of Shmoys and Tardos [100]). In particular, we can compute a $(1+\varepsilon)$-approximation for this instance in time $n^{O_\varepsilon(1)}$. One can show that the squares assigned to a block cell/subframe can be packed into them in a greedy manner, while losing only a factor of $1+\varepsilon$ in the objective. This yields a PTAS for the overall problem.

There are two crucial steps in which this algorithm needs a running time of $n^{O_\varepsilon(1)}$:

- guessing the large squares
- guessing the sizes of all cells and all subframes

In the next sections, we explain how we address these two problems. In section 7.2 we present a technique that allows us to guess the large squares in time $O_\varepsilon(1) \cdot n^{O(1)}$. Then in section 7.3 we present an EPTAS for the special case that $N \gg N'$, i.e., the knapsack is a rectangle with very large aspect ratio. In that case, there is a near-optimal packing with no large squares and only one cell that is elongated. This case will illustrate our techniques to address the second issue: to guess the sizes of all cells and frames. As we will see, this step will be interlaced with the selection of the squares and the computation of their packing. Building on these techniques, in section 7.4 we present our overall approach for the general case.

7.2 Guessing large squares faster

First, by losing at most a factor of $1 + O(\varepsilon)$ we round the profits of the squares to powers of $1 + \varepsilon$ in the range $[1, n/\varepsilon)$.

Lemma 22. *We can modify the input such that there are only $k^* = O(\frac{\log n}{\log(1+\varepsilon)})$ different square profits while losing at most a factor of $1 + O(\varepsilon)$.*

Proof. In a first step, we rescale the profits of the squares such that the square with largest profit has profit 1 and all profits lie in the interval $[0, 1]$. We now know that the optimal solution for this rescaled instance has profit at least 1. Therefore, we lose at most a factor $1 + \varepsilon$ if we remove all squares from the input that have profit less than ε/n, as these squares can never contribute more than εOPT to the solution. Finally, we multiply each profit by n/ε and hence end up with an instance where square profits are in the range $[1, n/\varepsilon]$. Losing another factor of $1+\varepsilon$, we round down the profit of each square to the next smaller powers of $1 + \varepsilon$. As a result, we obtain $k^* = \log_{1+\varepsilon}(n/\varepsilon) = O(\frac{\log n}{\log(1+\varepsilon)})$ different values for the square profits. □

As a result, we obtain k^* different values for the square profits and we define $I^{(k)} := \{i \in I | p_i = (1 + \varepsilon)^k\}$ for each $k = 0, ..., k^*$. This is a standard step that can be applied to any packing problem where we are guaranteed that $|\text{OPT}| \leq n$.

We would like to guess the large squares I'_L. Unfortunately, $|I'_L|$ can be as large as $F(\varepsilon) = (1/\varepsilon)^{2^{O(1/\varepsilon^4)}}$ so we cannot afford to enumerate all possibilities for I'_L. Also, for a square $i \in I$ it is a priori not even clear whether it appears as a large or small square in I' (some square in I'_L might even be smaller than some square in I'_S). To address this, first we guess $B := |I'_L|$; there are only $O_\varepsilon(1)$ possibilities. Then we do the following transformation of I'. If for a profit class $I^{(k)}$ we have that $|I' \cap I^{(k)}| \geq B/\varepsilon$ then we remove all squares in $I'_L \cap I^{(k)}$ from I'.

Lemma 23. *Suppose that $|I' \cap I^{(k)}| \geq B/\varepsilon$. Then $p(I'_L \cap I^{(k)}) \leq \varepsilon \cdot p(I' \cap I^{(k)})$.*

Proof. All squares in $I^{(k)}$ have the same profit and we have that $|I'_L| = B$, hence $|I'_L \cap I^{(k)}| \leq B$. Therefore, $|I'_S \cap I^{(k)}| \geq \frac{B(1-\varepsilon)}{\varepsilon}$ and $\frac{p(I'_L \cap I^{(k)})}{p(I' \cap I^{(k)})} \leq \frac{B}{B + \frac{B(1-\varepsilon)}{\varepsilon}} = \varepsilon$. □

Denote by I'' the resulting solution and let $I''_L := I'_L \cap (\bigcup_{k:|I' \cap I^{(k)}| < B/\varepsilon} I^{(k)})$. In I'' each profit class contributes at most B/ε squares in total or no large squares. This is useful since we can guess now the large squares in I'' fast. If a class $I^{(k)}$ contributes n_k squares to the solution we can assume w.l.o.g. that those are the n_k smallest squares of this class (since they are squares and all have roughly the same weight). Therefore, let $\tilde{I}$ denote the union of the B/ε smallest squares from each class $I^{(k)}$. We know that $|\tilde{I}| \leq O(\frac{\log n}{\log(1+\varepsilon)}) \cdot B/\varepsilon$ and that $I''_L \subseteq \tilde{I}$. We can now guess the correct choice of the at most B squares in I''_L in time $\binom{|\tilde{I}|}{B} \leq (\log_{1+\varepsilon}(n/\varepsilon) \cdot B/\varepsilon)^B = (\frac{\log n}{\log(1+\varepsilon)})^{F(\varepsilon)} \leq (\frac{1}{\log(1+\varepsilon)})^{O(F(\varepsilon))} \cdot n$. This culminates in the following lemma.

Lemma 24. *There are only $O_\varepsilon(1) \cdot n$ possibilities for the large squares I''_L in I''.*

7.3 Indirect guessing technique - special case

Suppose that the input knapsack has size $N \times N'$ with $N \geq \frac{1}{\varepsilon^4} N'$. For such an instance, there exists a $(1+\varepsilon)$-approximative solution which is a KR-packing (i.e., behaves like a single elongated cell) and there are no big squares [72].

From this packing, we first guess the number of frames and the number of subframes of each of them. There are $1/\varepsilon^{O(1/\varepsilon^4)}$ possibilities. For each subframe we guess whether it is a row subframe, a block subframe, or an empty subframe.

7.3.1 Guessing the block sizes

Then we guess the heights and widths of all block subframes, or *blocks* for short. This needs some preparation. Denote by K the number of blocks (which is implied by our previous guesses). We have that $K \leq O(1/\varepsilon^4)$ as each frame can contain at most one block subframe. Let α be the area of the block with largest area. We first guess an estimate of α. Intuitively, for doing this we use that α is by at most a factor $O(n)$ larger than the area of the largest square contained in the largest block.

Lemma 25. *We can compute a set of $T \leq \frac{n \log n}{\log(1+\varepsilon)}$ values $\alpha_1, ..., \alpha_T$ such that there exists an $i \in \{1, ..., T\}$ such that $\alpha_i \leq \alpha \leq (1+\varepsilon)\alpha_i$. Define $\tilde{\alpha} := \alpha_i$.*

Proof. Denote by $\hat{s}$ the size of the largest square packed in a block in the near-optimal solution we are considering. We have that $\hat{s}^2 \leq \alpha \leq n\hat{s}^2$. Therefore, consider the set

$$\left\{(1+\varepsilon)^j s_i^2 \quad | \quad \forall s_i \in I, \forall j \in \{0, \dots, \log_{1+\varepsilon} n\}\right\}.$$

It contains $\log_{1+\varepsilon} n$ values between s_i^2 and ns_i^2 for every square size s_i, hence at most $n \log_{1+\varepsilon} n$ values in total. As the values are by at most a factor $1+\varepsilon$ apart, the claim follows by choosing α_i to be the largest value in this set below α. □

We guess $\tilde{\alpha}$ in time $n \log n / \log(1+\varepsilon)$.

We will now prove the following useful lemma, which we will use at various places:

Lemma 26. *Let R be a rectangle of side lengths a, b and S a set of squares of total profit $p(S)$ such that each of the squares in S has side length at most $\varepsilon \min\{a, b\}$ and the total area of S is $a(S) \leq ab$. Then we can pack a subset $S' \subseteq S$ with total profit $p(S') \geq (1 - O(\varepsilon))p(S)$ into R.*

Proof. Each square has area at most $\varepsilon^2 \min\{a, b\}^2 \leq \varepsilon^2 ab$. If $a(S) \leq ab(1-2\varepsilon)^2$, then $a(S) \leq a(1-\varepsilon)b(1-2\varepsilon) \leq (a-\varepsilon)(b-2\varepsilon)$ and hence by lemma 20 we can pack all squares from S into the rectangle R. Therefore, assume that $a(S) > ab(1-2\varepsilon)^2$, and try to find a subset of squares from S that has small profit (at most $O(\varepsilon)p(S)$) such that when we remove these squares from S, the total area drops below the bound from lemma 20. We assign the squares to groups such that each group contains squares with a total area between $4\varepsilon ab$ and $(4\varepsilon + \varepsilon^2)ab$ (we can achieve this by a simple greedy algorithm: assign squares into one group until the total area of squares in this group is at least $4\varepsilon ab$; the upper bound follows from the upper bound on the square size). We get that $\frac{ab(1-2\varepsilon)^2}{(4\varepsilon+\varepsilon^2)ab} = \Omega(1/\varepsilon)$, and thus there is one group j whose squares S_j have a total profit of at most $O(\varepsilon)p(S)$. The squares in this group have an area of at

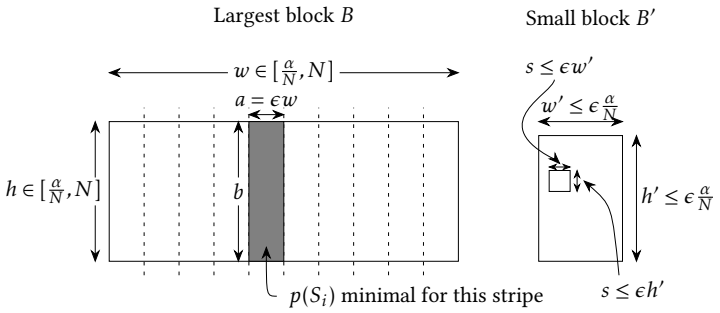


Figure 7.3: Illustration for proof of lemma 27.

least $4\varepsilon ab \geq (1-(1-2\varepsilon)^2)ab$ and hence the squares in $S \setminus S_j$ have total area at most $ab - (1-(1-2\varepsilon)^2)ab = (1-2\varepsilon)^2 ab$ and we can pack them using lemma 20. □

Next, we simplify the packing by removing blocks that are very small.

Lemma 27. *By losing a factor* $1 + O(\varepsilon)$ *we can remove all blocks with area less than* $\tilde{\alpha}\frac{\varepsilon}{KN}$.

Proof. Consider the largest block (according to area) B, let w be its width and h its height and suppose that $w \geq h$. Define $1/\varepsilon$ vertical strips of equal width in this cell, and let $p(S_i)$ ("the profit of strip S_i") denote the total profit of all squares intersecting the i-th strip. A square cannot intersect more than two strips, as the square size is at most εw and this is also the width of the strips. This means that $\sum_{i=1}^{1/\varepsilon} p(S_i) \leq 2\sum_{i \text{ packed in } B} p_i = 2p(B)$ (where $p(B)$ denotes the total profit of squares in block B). Consider the strip with smallest profit. It must have total profit $p(S_i) \leq 2\varepsilon p(B)$. Hence, we can remove all squares touching the strip with lowest $p(S_i)$ at cost of at most an $O(\varepsilon)$ fraction of the profit of all squares in this cell. See fig. 7.3 for illustration.

The free strip has width $a := \varepsilon w$ and height $b := h$ and thus an area of $\alpha^* := \varepsilon hw = \varepsilon\alpha$, and we now pack the squares of the small blocks (i.e. blocks with area less than $\varepsilon\frac{\tilde{\alpha}}{KN}$) into this strip using NFDH. Let S be the set of these squares. The squares in S have a total area of at most $\varepsilon\frac{\tilde{\alpha}}{N} \leq \varepsilon\frac{\alpha}{N} = \alpha^*$.

Let now B' be one of the small blocks with width w' and height h', and let s be the size of a square in this block. We want to show that $s \leq \varepsilon \min\{a, b\}$, and then argue that this gives us the desired result. We know from the definition of small blocks that $h'w' \leq \varepsilon\frac{\alpha}{KN}$, and as $w', h' \geq 1$, we know that both h' and w' must be at most $\varepsilon\frac{\alpha}{KN} \leq \varepsilon\frac{\alpha}{N}$. We know $\min\{a, b\} \geq \varepsilon\frac{\alpha}{N}$ (as again $\max\{a, b\} \leq N$ and $ab = \varepsilon\alpha$) and $s \leq \varepsilon w'$, and thus $s \leq \varepsilon w' \leq \varepsilon^2\frac{\alpha}{N} \leq \varepsilon \min\{a, b\}$.

Now consider lemma 26. In our case, the rectangular area in which we want to pack the squares from small blocks has side lengths $a = \varepsilon w$ and $b = h$. These squares have size at most $\varepsilon \min\{a, b\}$ and total area at most that of the free area, as established above. Hence, NFDH will pack a subset of S of profit at least $(1 - O(\varepsilon))p(S)$ into the largest block by lemma 26. □

Using that our input squares are squares and that no cell can contain more than n squares we can prove the following lemma.

Lemma 28. *For each block of width $w_{C,\ell,j}$ and height $h_{C,\ell,j}$ we can assume that*

$$h_{C,\ell,j}, w_{C,\ell,j} \in \left[\sqrt{\frac{\varepsilon\tilde{\alpha}}{nKN}}, \sqrt{(1+\varepsilon)\tilde{\alpha}n}\right].$$

Proof. Consider a block B of height and width h, w. Let $\hat{s}_i$ be the size of the largest square in this block, then we have that $h, w \geq \hat{s}_i$ (as this square must fit in the block). We can assume that $h \leq n\hat{s}_i \leq nw$ and $w \leq n\hat{s}_i \leq nh$ (since at most n squares can be packed in this box and each of them has size at most $\hat{s}_i$ by definition of $\hat{s}_i$), and hence $w \geq h/n$, which finally gives $\frac{h^2}{n} \leq w \cdot h \leq (1+\varepsilon)\tilde{\alpha} \Leftrightarrow h \leq \sqrt{(1+\varepsilon)\tilde{\alpha}n}$ and $\varepsilon\frac{\tilde{\alpha}}{KN} \leq wh \leq nh^2 \Leftrightarrow h \geq \sqrt{\tilde{\alpha}\frac{\varepsilon}{nKN}}$ and therefore we get $h \in [\sqrt{\frac{\tilde{\alpha}\varepsilon}{nKN}}, \sqrt{(1+\varepsilon)\tilde{\alpha}n}]$ (the claim for w follows analogously). □

This ensures that there is a polynomial range for the sizes of the cells. Next, we show that we can round down the side lengths of the blocks.

Lemma 29. *By losing a factor $1 + O(\varepsilon)$ we can round down the height and width of each block such that it becomes a power of $1 + \varepsilon$.*

Proof. We down round each block height and width to the nearest power of $1 + \varepsilon$. This results in $O(\log_{1+\varepsilon} n)$ different values. Now, to show that we can still pack the squares into the blocks without losing too much profit, we do the following: Let B be some block of width w_B and height h_B and assume w.l.o.g. that $w_B \leq h_B$; denote by w'_B and h'_B the rounded side lengths of B. Assume $w'_B = (1+\varepsilon)^j$ and $h'_B = (1+\varepsilon)^i$, then we know that $w_B < (1+\varepsilon)^{j+1}$ and $h_B < (1+\varepsilon)^{i+1}$. We divide B into $\frac{1}{2\varepsilon}$ many strips of equal width $2\varepsilon h_B$. Each square touches at most two strips, so there is a strip s.t. all squares who touch this strip have total profit at most $O(\varepsilon p_B)$ (where p_B is the total profit of squares in block B). We remove the squares from this strip and hence gain some free space of area $2\varepsilon h_B w_B$. The squares touching the area that we lose due to shrinking have total area at most $(h_B - h'_B)w_B + (w_B - w'_B)h_B \leq ((1+\varepsilon)^{i+1} - (1+\varepsilon)^i)w_B + ((1+\varepsilon)^{j+1} - (1+\varepsilon)^j)h_B \leq \varepsilon h'_B w_B + \varepsilon w'_B h_B \leq 2\varepsilon w_B h_B$. Hence, we can pack a subset of these squares into the free area using NFDH (see lemma 26) and lose only an ε fraction of the profit. □

After applying lemma 29 for each of the rounded values there are only $O\left(\frac{\log n}{\log(1+\varepsilon)}\right)$ possibilities left. Thus, we can guess *all* these values in running time $\left(\frac{\log n}{\log(1+\varepsilon)}\right)^{O(1/\varepsilon^4)}$.

In the special case studied in this section all squares i with $s_i \leq \varepsilon^4 \cdot w_C$ are packed into the blocks. We call them *small squares*. All squares i with $s_i > \varepsilon^4 \cdot w_C$ are packed into the rows, we call them the *row squares*. Each edge of each block is by a factor $\Omega(1/\varepsilon^2)$ larger than the size of any small square by the definition of KR-packings. Thus, we can find a $(1+\varepsilon)$-approximation for the small squares using greedy algorithms for selecting and packing the squares.

Lemma 30. *There is a $(1+\varepsilon)$-approximation algorithm with a running time of $n^{O(1)}$ for computing the most profitable set of small squares that can be packed into the blocks.*

Proof. According to lemma 26, it suffices to select some set of small squares S that has total area at most that of the blocks. The condition that squares are by a factor ε^2 smaller than the blocks is already established by the definition of KR-packings (the

squares have $s_i \le \varepsilon^4 h_C$ while the width and height of the block are at least $\varepsilon^2 h_C$). We do the selection of squares by solving a simple Knapsack problem that selects the most-profitable set of squares whose total area does not exceed the blocks' total area (for this problem, an FPTAS exists, see [69]). □

7.3.2 Packing row subframes via indirect guessing

The packing decision for the row squares is more challenging. First, for each subframe $F_{C,\ell,j}$ we guess the value $f(C,\ell,j)$. For each row subframe $F_{C,\ell,j}$ denote by $\overline{w}_{C,\ell,j}$ the number of squares packed in $F_{C,\ell,j}$. Observe that they all fit into $F_{C,\ell,j}$ even if each of them has a width of up to $k_{f(C,\ell,j)}$ since $w_{C,\ell,j} \ge \overline{w}_{C,\ell,j} \cdot k_{f(C,\ell,j)}$. As the next lemma shows we can guess the values $\overline{w}_{C,\ell,j}$ approximately by, intuitively, allowing only powers of $1+\varepsilon$ for them.

Lemma 31. *There is a set L with $|L| = O\left(\frac{\log n}{\log(1+\varepsilon)}\right)$ such that by losing a factor $1+O(\varepsilon)$ we can assume that $\overline{w}_{C,\ell,j} \in L$ for each subframe $F_{C,\ell,j}$.*

Proof. We can assume that for each C,ℓ,j we have that $1 \le \overline{w}_{C,\ell,j} \le n$. We define $L := \left\{ \lfloor (1+\varepsilon)^i \rfloor \;\middle|\; \forall i \ge 0, (1+\varepsilon)^i \le n \right\}$ and thus $|L| = O(\log_{1+\varepsilon} n) = O\left(\frac{\log n}{\log(1+\varepsilon)}\right)$.

Now, we round down each value $\overline{w}_{C,\ell,j}$ to the nearest value in L. Consider one such value $\overline{w}_{C,\ell,j}$ and let $\widetilde{w}_{C,\ell,j} = (1+\varepsilon)^i$ be the new (smaller) value. When we do this, we might lose at most $\overline{w}_{C,\ell,j} - \widetilde{w}_{C,\ell,j} \le (1+\varepsilon)^{i+1} - (1+\varepsilon)^i = \varepsilon(1+\varepsilon)^i = \varepsilon \widetilde{w}_{C,\ell,j} \le \varepsilon \overline{w}_{C,\ell,j}$ many squares. For these, we pick the ones with smallest profit in $F_{C,\ell,j}$, so we lose profit of at most an ε fraction of the total profit of the squares packed in $F_{C,\ell,j}$. □

We guess all values $\overline{w}_{C,\ell,j}$ in time $\left(\frac{\log n}{\log(1+\varepsilon)}\right)^{O(F(\varepsilon))} = O_\varepsilon(1) \cdot n^{O(1)}$. Ideally, we would like to guess the values k_j, which give us the heights of the row subframes. Then it would be very easy to compute the packing: for each index r the squares i with $s_i \in (k_{r-1}, k_r]$ form a group I_r. In the packing that we aim at (see lemma 21) each row of height k_r contains only squares from I_r. We already guessed the total number of squares in each row and this yields the total number of squares from each group I_r. Denote by n_r this value, i.e., $n_r := \sum_\ell \sum_{j:f(C,\ell,j)=r} \overline{w}_{C,\ell,j}$. We simply select the n_r most profitable squares from I_r and pack them greedily into the rows of height k_r.

Unfortunately, we cannot guess the values k_j directly, as we have $O(n)$ candidate values for those (the sizes of the input squares) and we have $O_\varepsilon(1)$ of these values. Instead, we guess approximations for them *indirectly*. Recall that $k_0 = \varepsilon^4 \cdot w_C$. We want to guess a value k_1' with $k_1' \le k_1$ such that if we define the first group to be $I_1' := \left\{ i \mid s_i \in (k_0, k_1'] \right\}$ (instead of $I_1 := \{ i \mid s_i \in (k_0, k_1] \}$) then the greedy packing of a subset of the squares in this group yields a profit that is almost as large as $p(\mathrm{OPT} \cap I_1)$. To this end we ask ourselves: how much profit could we obtain from group I_1' if we defined I_1' to contain all squares in the size range $(k_0, x]$? We know that in the KR-packing the squares from group I_1 are packed only into the rows of height k_1 and we know already that there are n_1 slots for them.

We want to define k_1', then $I_1' := \{ i \mid s_i \in (k_0, k_1'] \}$, and then pack the n_1 most profitable squares from I_1' into the n_1 slots in the rows of height k_1 (strictly speaking we make the rows less high such that they have height $k_1' \le k_1$).

We want to allow only few candidate values for k_1'. To this end, we define a function $\overline{p}_1(x)$ where each value x is mapped to the total profit of the n_1 most profitable squares i with $s_i \in (k_0, x]$, i.e., the total profit of the first n_1 squares from

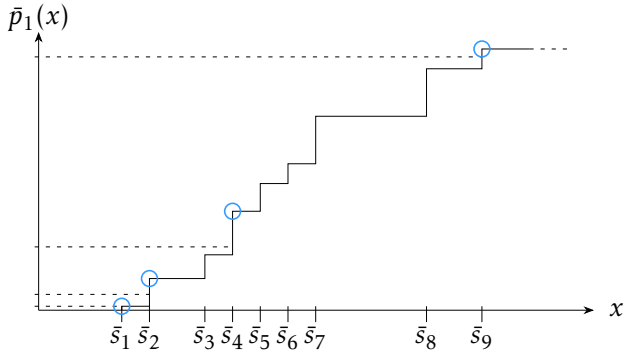


Figure 7.4: Example how the function $\overline{p}_1(x)$ might look like. $\bar{s}_1,\ldots,\bar{s}_9$ denote the distinct square sizes in increasing order. The graph shows the function $\overline{p}_1(x)$. Dashed lines denote powers of $(1+\varepsilon)$, and the points highlighted with blue circles denote which values (i.e. which $\bar{s}_i$'s) are under consideration for k_1'.

group I_1' if we defined $I_1' := \{i \mid s_i \in (k_0, x]\}$. Note that $\overline{p}_1$ is a non-decreasing step-function with at most n steps. Due to the rounding of the square profits, we know that $\max_x \overline{p}_1(x) \leq n^2/\varepsilon$. Now instead of allowing all possible values (all input item sizes) for k_1', we allow only those values x where $\overline{p}_1(x)$ increases by a power of $1+\varepsilon$, i.e., such that there is a $\kappa \in \mathcal{N}$ with $(1+\varepsilon)^\kappa \in (\overline{p}_1(x-\delta), \overline{p}_1(x)]$ for each $\delta > 0$. For an illustration, see fig. 7.4. As a result, there are only $O(\log n/\log(1+\varepsilon))$ such values and we define one of them to be k_1' such that our profit from the resulting group $I_1' := \{i \mid s_i \in (k_0, k_1']\}$ will be at least $\frac{1}{1+\varepsilon} \cdot p(\mathrm{OPT} \cap I_1)$ and $k_1' \leq k_1$. The latter inequality is important because our rows must fit into C, i.e., the rows and blocks intersected by each vertical line must not have a total height of more than h_C. Thus, we guess k_1' in time $O(\log n/\log(1+\varepsilon))$.

Lemma 32. *Given $\overline{p}_1$ we can compute a set of $O(\log n/\log(1+\varepsilon))$ values which contains a value $k_1' \leq k_1$ such that $\overline{p}_1(k_1') \geq \frac{1}{1+\varepsilon}\overline{p}_1(k_1) = \frac{1}{1+\varepsilon}p(\mathrm{OPT} \cap I_1)$.*

Proof. Let $\bar{s}_1,\ldots,\bar{s}_{n'}$ denote the distinct sizes of large squares (i.e. squares with size $> k_0$). We compute $\overline{p}_1$. Note that $\overline{p}_1$ is a step function having all its steps at points x such that $x = \bar{s}_i$ for some square i, as only for these values of x new squares become part of the set I_1 (there does not have to be a step at such a point x; e.g., consider the case that the new squares of size x have very little profit and therefore they are not included in the subset of the n_1 most profitable squares of I_1). Intuitively, we define C to be those points where the function $\overline{p}_1$ "jumps" over a value $(1+\varepsilon)^j$ for the first time. Formally, let $C := \left\{\bar{s}_i \mid \exists j : \overline{p}_1(\bar{s}_{i-1}) < (1+\varepsilon)^j \leq \overline{p}_1(\bar{s}_i)\right\}$ where we define for convenience $\overline{p}_1(\bar{s}_0) = 0$. We have $|C| \leq \log_{1+\varepsilon} \frac{n^2}{\varepsilon} = O\left(\frac{\log n}{\log(1+\varepsilon)}\right)$.

Now, let k_1' be the largest value in C that is not larger than k_1. Then there is a j s.t. $\overline{p}_1(k_1') \geq (1+\varepsilon)^j$ and $\overline{p}_1(k_1) < (1+\varepsilon)^{j+1}$ and thus $\overline{p}_1(k_1') \geq \frac{1}{1+\varepsilon}\overline{p}_1(k_1)$. □

We iterate and we want to guess a value k_2' (instead of k_2). Recall that there are n_2 slots for the group I_2 and in the optimal packing we select the n_2 most profitable squares in I_2. We want to define k_2' such that $k_2' \leq k_2$ and such that if we pack the n_2 most profitable squares from the group $I_2' := \{i \mid s_i \in (k_1', k_2']\}$ then we obtain almost the same profit as $p(\mathrm{OPT} \cap I_2)$. We use the same approach as for defining k_1'. We

define a function $\overline{p}_2(x)$ that maps each value x to the profit of the n_2 most profitable squares i with $s_i \in [k_1', x)$. Observe that $\overline{p}_2(k_2) \geq p(\mathrm{OPT} \cap I_2)$ because $k_1' \leq k_1$. Like before, for defining k_2' we allow only values x where $\overline{p}_2(x)$ increases by a factor of $1+\varepsilon$. There are only $O(\log n/\log(1+\varepsilon))$ such values.

Lemma 33. *Given $\overline{p}_2$ we can compute a set of $O(\log n/\log(1+\varepsilon))$ values which contains a value $k_2' \leq k_2$ such that $\overline{p}_2(k_2') \geq \frac{1}{1+\varepsilon}\overline{p}_2(k_2) \geq \frac{1}{1+\varepsilon}p(\mathrm{OPT} \cap I_2)$.*

Proof. The argumentation is basically analogous to the proof of the previous lemma, however, there is one difference. As $I_2' = \{i \mid s_i \in (k_1', k_2']\}$, this set also depends on the previous guess k_1'. We define the set C as before and let k_2' be the largest value in C that is not larger than k_2 as before. As $k_1' \leq k_1$, we know that the set $\{i \mid s_i \in (k_1', k_2']\}$ is a superset of $\{i \mid s_i \in (k_1, k_2']\}$, therefore the argumentation carries over. □

We guess the value k_2' in time $O(\log n/\log(1+\varepsilon))$ and we define $I_2' := \{i \mid s_i \in (k_1', k_2']\}$. We continue in this way until for each j we guessed a value k_j' and the corresponding set $I_j' := \{i \mid s_i \in (k_{j-1}', k_j']\}$. Since there are $1/\varepsilon^4$ such values and for each of them there are $O(\log n/\log(1+\varepsilon))$ possibilities, this yields $(\log n)^{O_\varepsilon(1)}$ possibilities in total. Observe that each function $\overline{p}_j$ depends on the previous guess k_{j-1}'. There are $(\log n)^{O_\varepsilon(1)}$ guesses overall and we can enumerate all of them in time $O_\varepsilon(n) \cdot (\log n)^{O_\varepsilon(1)}$.

The values k_j' imply the height of each row: for each row subframe $F_{C,\ell,j}$ we define its height $h_{C,\ell,j}$ to be $k'_{f(C,\ell,j)}$. In case that one frame $F_{C,\ell}$ is higher than h_C, i.e., $\sum_j h_{C,\ell,j} > h_C$, then we reject this set of guesses. In particular, then there must be some value k_j' with $k_j' > k_j$ and thus our guess was wrong. Among all remaining guesses, we select the one that yields the maximum total profit if we take the n_r most profitable squares from each group I_r'.

Theorem 8. *There is a $(1+\varepsilon)$-approximation algorithm for the two-dimensional knapsack problem for squares if the height and the width of the knapsack differ by more than a factor of $1/\varepsilon^4$. The running time of the algorithm is $O_\varepsilon(1) \cdot n^{O(1)}$.*

Proof. The algorithm proceeds as follows: We first guess the number of frames and the number of subframes of each frame in time $1/\varepsilon^{O(1/\varepsilon^4)} = O_\varepsilon(1)$. Then we guess $\tilde{\alpha}$ in time $\frac{n \log n}{\log(1+\varepsilon)} = O_\varepsilon(1) \cdot n^{O(1)}$ and use it to guess the rounded width and height of each of the $O(1/\varepsilon^4)$ blocks in time $(\frac{\log n}{\log(1+\varepsilon)})^{O(1/\varepsilon^4)} = O_\varepsilon(1) \cdot n^{O(1)}$. Row squares can be packed into the corresponding row subframes in a greedy manner and are guaranteed to fit. We can use an FPTAS for the knapsack problem to select a set of small items whose total area does not exceed the area of the blocks while maximizing their total profit. Using lemma 26, we can pack almost all of these into the blocks while losing at most an ε fraction of the profit. The total running time is $O_\varepsilon(1) \cdot n^{O(1)}$. □

7.4 Indirect guessing technique - general case

We present our EPTAS for the general case now. There are two new difficulties compared to the special case from section 7.3:

- There can be several elongated cells now (rather than only one that spans the entire knapsack). For elongated cells C with $w_C \geq h_C$ we cannot guess the height in time $O_\varepsilon(1) \cdot n^{O(1)}$ and for elongated cells C' with $h_{C'} \geq w_{C'}$ we cannot guess the width in time $O_\varepsilon(1) \cdot n^{O(1)}$. Therefore, we incorporate the guessing of these values into the indirect guessing framework for the heights of the horizontal (widths of the vertical) row subframes.
- We can no longer partition the input squares into two groups such that the squares from one group are assigned only to the blocks (like the squares i with $s_i \leq \varepsilon^4 \cdot h_C$ in the previous special case) and the others are only assigned to the rows (like the squares i with $s_i > \varepsilon^4 \cdot h_C$). Therefore, for each group I_r we guess an estimate of either the total number of squares in each block or the total occupied area in each block in order to apply our framework for indirect guessing.

First, we guess the large squares I'_L as described in section 7.2. Next, we guess the number of block and elongated cells, and for each elongated cell the number of frames and the number of subframes in each frame as well as the type of each subframe (block subframe, row subframe, empty subframe). We will use the term *block* for a block cell or a block subframe. We guess the sizes of all blocks like in section 7.3, with the only difference that we can now have up to $O(F(\varepsilon))$ many blocks instead of $O(1/\varepsilon^4)$: we first guess the value $\tilde{\alpha}$ in time $O(\frac{n \log n}{\log(1+\varepsilon)})$ and then the heights and widths of all blocks in time $(\frac{\log n}{\log(1+\varepsilon)})^{O(F(\varepsilon))}$.

Lemma 34. *By losing a factor $1 + O(\varepsilon)$ we can round the heights and widths of all block cells and all block subframes such that we can guess all of them in time $\frac{n \log n}{\log(1+\varepsilon)} \cdot (\frac{\log n}{\log(1+\varepsilon)})^{O(F(\varepsilon))} \leq O_\varepsilon(1) \cdot n^{O(1)}$.*

In the packing given by lemma 21 each elongated cell is packed as a KR-packing. Therefore, for each horizontal elongated cell C there is a set of at most $1/\varepsilon^4$ values $k_0, k_1, k_2, \ldots$ defined *locally* for C, such that for the height of each row subframe $F_{C,\ell,j}$ of C there is an integer $f(C,\ell,j) \geq 1$ such that $h_{C,\ell,j} = k_{f(C,\ell,j)}$ and $F_{C,\ell,j}$ contains only squares i with $s_i \in (k_{f(C,\ell,j)-1}, k_{f(C,\ell,j)}]$, while we assume that $k_0 = 0$. We establish now that we can assume that there are $O_\varepsilon(1)$ *global* values with these properties.

Lemma 35. *Let $k_0 = 0$. By losing a factor $1 + O(\varepsilon)$ and by increasing the number of elongated cells within the knapsack and the total number of row subframes in all elongated cells to $O(1/\varepsilon^{10} F(\varepsilon)^3)$ each we can assume that there are at most $O(1/\varepsilon^4 F(\varepsilon)^2)$ (global) values $k_1 \leq k_2 \leq \ldots$ such that for each horizontal (or vertical) row subframe $F_{C,\ell,j}$ there is a value $f(C,\ell,j) \geq 1$ such that $h_{C,\ell,j} = k_{f(C,\ell,j)}$ (or $w_{C,\ell,j} = k_{f(C,\ell,j)}$) and $F_{C,\ell,j}$ contains only squares i with $s_i \in (k_{f(C,\ell,j)-1}, k_{f(C,\ell,j)}]$. Furthermore, for each block B of height h_B and width w_B and each r, we know that either $k_r \leq \varepsilon^2 \min\{w_B, h_B\}$ or $k_{r-1} \geq \varepsilon^2 \min\{w_B, h_B\}$.*

Proof. Let $C_1, C_2, \ldots$ be the elongated cells. Denote by $k_1^{(i)}, k_2^{(i)}, \ldots$ the local k_r-values for cell C_i. We define the set

$$K = \left\{ k_j^{(i)} \;\middle|\; \forall i, j \right\} \cup \left\{ \varepsilon^2 \min\{w_B, h_B\} \;\middle|\; \forall \text{ blocks } B \text{ of width } w_B \text{ and height } h_B \right\}.$$

Clearly, $|K| \leq 1/\varepsilon^4 \cdot F(\varepsilon) + F(\varepsilon) = O(1/\varepsilon^4 F(\varepsilon)^2)$.

It remains to show that there is a KR-packing that corresponds to these global k-values. Consider the original KR-packing of one elongated cell, i.e. the one using local k_r-values. We will show how to transform it into another KR-packing that adheres to the global values K. Consider a row subframe $F_{C,\ell,j}$ of height $k_r^{(i)}$. Assume that the largest value smaller than $k_r^{(i)}$ in K, k^*, is a value that does not belong to the local k_r-values of this cell (it might be some $k_{r'}^{(i')}$ for $i' \neq i$ or a value $\varepsilon^2 \min\{w_B, h_B\}$); so in particular $k^* \in (k_{r-1}^{(i)}, k_r^{(i)})$. The subframe $F_{C,\ell,j}$ might now contain squares of side length in $(k_{r-1}^{(i)}, k_r^{(i)}]$, i.e. in particular squares with size smaller than k^* and also squares with size larger than k^*.

In a KR-packing that adheres to the values K, these squares must not be packed in the same row subframe. We therefore do the following: Assume w.l.o.g. that the squares packed in subframe $F_{C,\ell,j}$ are sorted in decreasing order of size from left to right. We split this one elongated cell into several smaller ones: the first one contains all frames above the "problematic" frame $F_{C,\ell}$, the second contains all frames below $F_{C,\ell}$, the third consists of one frame that has subframes $F_{C,\ell,1}, \ldots, F_{C,\ell,j-1}$, the fourth ones contains subframes $F_{C,\ell,j+1}, \ldots, F_{C,\ell,p}$ (where p is the number of subframes of $F_{C,\ell}$) contained in one single frame and the fifth consists only of the subframe $F_{C,\ell,j}$ (see fig. 7.5 for illustration). Note that all these are valid KR-packings. Now, in the new elongated cell that only contains subframe $F_{C,\ell,j}$, we can split the frame into two frames, each having one subframe: one subframe has height $k_r^{(i)}$ and contains only squares of size in $(k^*, k_r^{(i)}]$, the other one has height k^* and contains squares of size in $(k_{r-1}^{(i)}, k^*]$.

The number of row subframes before this splitting of elongated cells was at most $O(F(\varepsilon)) \cdot O(1/\varepsilon^6)$. In the worst case, we splitted each row subframe into $O(1/\varepsilon^4 F(\varepsilon)^2)$ row subframes, hence after this process we have at most $O(1/\varepsilon^{10} F(\varepsilon)^3)$ many row subframes. As each elongated cell must contain at least one row subframe or block subframe and the number of block subframes stays unchanged, we have at most $O(1/\varepsilon^{10} F(\varepsilon)^3)$ many elongated cells as well.

By introducing k_r-values of the form $\varepsilon^2 \min\{w_B, h_B\}$ for each block B, we also made sure that for each B and $k_r \in K$, it either holds that $k_r \leq \varepsilon^2 \min\{w_B, h_B\}$ or $k_{r-1} \geq \varepsilon^2 \min\{w_B, h_B\}$. □

In the next step, we guess some quantities for the row subframes (similar to section 7.3).

Lemma 36. *By losing a factor $1 + O(\varepsilon)$ in time $O_\varepsilon(1) \cdot n$ we can guess for each row subframe $F_{C,\ell,j}$*

- *the number $\overline{w}_{C,\ell,j}$ of squares packed in $F_{C,\ell,j}$,*
- *the value $f(C, \ell, j)$ indicating that if $F_{C,\ell,j}$ is a horizontal row then for its height $h_{C,\ell,j}$ it holds that $h_{C,\ell,j} = k_{f(C,\ell,j)}$, and if $F_{C,\ell,j}$ is a vertical row then for its width $w_{C,\ell,j}$ it holds that $w_{C,\ell,j} = k_{f(C,\ell,j)}$*

Proof. First, we want to guess the values $\overline{w}_{C,\ell,j}$ for each row subframe $F_{C,\ell,j}$. Similar to lemma 31, we round down $\overline{w}_{C,\ell,j}$ to the nearest value in L (as defined in the proof of lemma 31), and by throwing away the least profitable squares we lose at most an ε fraction of the total profit in $F_{C,\ell,j}$. For each row subframe, we have $O\left(\frac{\log n}{\log(1+\varepsilon)}\right)$

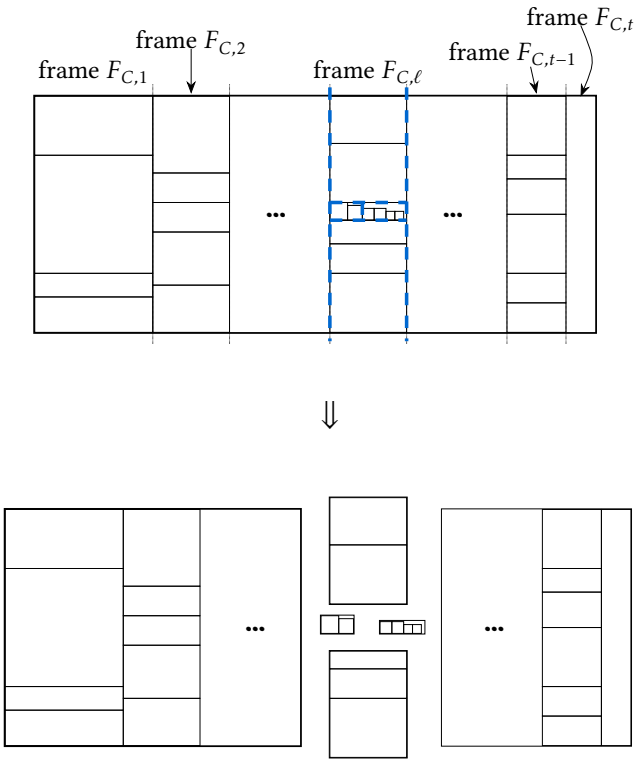


Figure 7.5: Splitting one elongated cell into five.

possible values, i.e., we can guess the $\overline{w}_{C,\ell,j}$-values for all row subframes in time $\left(\frac{\log n}{\log(1+\varepsilon)}\right)^{O(1/\varepsilon^{10}F(\varepsilon)^3)} = (\log n)^{O_\varepsilon(1)} \leq O_\varepsilon(1) \cdot n$.

Second, we need to guess $f(C,\ell,j)$ for each of the at most $O(1/\varepsilon^{10}F(\varepsilon)^3)$ row subframes. There are $O(1/\varepsilon^4 F(\varepsilon)^2)$ many possibilities per subframe, so there are at most $(1/\varepsilon^4 \cdot F(\varepsilon)^2)^{O(1/\varepsilon^{10}F(\varepsilon)^3)}$ possibilities in total, which is a constant only depending on ε.

Hence, in total we need time $O_\varepsilon(1) \cdot n$ to guess the values $\overline{w}_{C,\ell,j}$ and $f(C,\ell,j)$ for all row subframes. □

As before, for each r denote by n_r the total number of squares in rows of height k_r, i.e., $n_r := \sum_C \sum_\ell \sum_{j:f(C,\ell,j)=r} \overline{w}_{C,\ell,j}$.

Note that we did not guess the heights and widths for the row subframes. Those are implied once we guess the values $k_1, k_2, \ldots$ since the height/width of each vertical/horizontal row subframe $F_{C,\ell,j}$ is at least $\overline{w}_{C,\ell,j} \cdot k_{f(C,\ell,j)}$ and a larger height/width is not necessary to fit $\overline{w}_{C,\ell,j}$ squares of size at most $k_{f(C,\ell,j)}$. In contrast to section 7.3 a square i with size $i \in (k_r, k_{r+1}]$ might be assigned to a row (of height k_{r+1}) *or* to a block. In our indirect guessing framework, we need in particular a function $\overline{p}_1(x)$ that maps each x to the profit we would obtain from group I_1 if the group I_1 contained all squares i with $s_i \in (k_0, x]$. This profit depends now not only on the number of slots in the rows for the squares in I_1 but also in the space that we allocate to I_1 in each block. In order to handle this, we first guess for each k_r-value for $r \geq 2$, whether $\frac{k_r}{k_{r-1}} > 1+\varepsilon$ or $\frac{k_r}{k_{r-1}} \leq 1+\varepsilon$. We denote by K_L the set $\left\{k_r \mid \frac{k_r}{k_{r-1}} > 1+\varepsilon\right\} \cup \{k_1\}$ of all

k_r-values for which the first inequality holds (plus k_1) and by K_S all values with the latter property, i.e., $\left\{k_r \mid \frac{k_r}{k_{r-1}} \leq 1+\varepsilon\right\}$. If $k_r \in K_L$ then we guess for the set I_r and each block B the area that squares from I_r occupy in B. Otherwise, if $k_r \in K_S$ then for each block B we guess the number of squares from I_r packed in B. Recall that for each block B we already guessed its height and width.

Lemma 37. *We can compute for each k_r two sets L'_r, L''_r which satisfy $|L'_r|, |L''_r| = O(\log n / \log(1+\varepsilon))$ and such that by losing a factor $1+O(\varepsilon)$, we can assume that for each block B and each group $I_r = \{i \mid s_i \in (k_{r-1}, k_r]\}$ the following holds:*

- *If $k_r \in K_L$, then there is a value $a_{B,r} \in L'_r$ such that the total area used by squares of group I_r in B is bounded by $a_{B,r}$ and either $a_{B,r} = 0$ or $a_{B,r} k_r^2 \geq \frac{1}{\varepsilon} k_r^2$.*
- *If $k_r \in K_S$, then there is a value $n_{B,r} \in L''_r$ such that the number of squares of group I_r packed in B is bounded by $n_{B,r}$.*

Moreover, for each block B we have $\sum_{r \in K_L} a_{B,r} + \sum_{r \in K_S} n_{B,r} k_r^2 \leq (1+\varepsilon) a_B$ where a_B denotes the area of B.

Proof. We will prove the two statements of the lemma separately.

Case 1: $k_r \in K_L$ Let $L'_r = \left\{(1+\varepsilon)^i \;\middle|\; 1/\varepsilon \leq (1+\varepsilon)^i \leq n\right\} \cup \{0\}$. Clearly, the size of L'_r is at most $\log_{1+\varepsilon}(n) + 1 = O(\frac{\log n}{\log(1+\varepsilon)})$. Consider a block B and a group I_r; let $A^*_{B,r}$ be the area of squares of I_r in B in the solution under consideration and let $a^*_{B,r} := A^*_{B,r}/k_r^2$. At a first glance, we would want to guess a value $a_{B,r}$ such that $A_{B,r} := a_{B,r} \cdot k_r^2 \leq A^*_{B,r}$ and most of the squares of I_r still fit in an area of size $A_{B,r}$ (most of the squares means enough squares so we do not lose too much profit). However, we will later see why this might not be good enough. Thus, we will make a case distinction on $A^*_{B,r}$. Let L^* be the largest value in L'_r that is at most $A^*_{B,r}/k_r^2$.

First, assume that $L^* \geq 1/\varepsilon$. In that case, define $a_{B,r} := L^*$, and thus $A_{B,r} \geq 1/\varepsilon k_r^2$. Say $A_{B,r} = (1+\varepsilon)^i k_r^2$, thus we know $A^*_{B,r} < (1+\varepsilon)^{i+1} k_r^2$ and therefore $A^*_{B,r} - A_{B,r} < \left((1+\varepsilon)^{i+1} - (1+\varepsilon)^i\right) k_r^2 = \varepsilon(1+\varepsilon)^i k_r^2 = \varepsilon A_{B,r}$. That is, we lose some area by our guess of the area for items of group r in B, but this "lost" area is at most an ε fraction of the guessed area. Hence, we can do the following: Consider the squares of I_r packed in B in any order. We want to create buckets containing squares of total area $\varepsilon A_{B,r}$, therefore, add squares to the first bucket until the total area reaches $\varepsilon A_{B,r}$, possibly including one square only partially. Continue to create buckets this way, starting by adding the remaining partial square (if existing). We get $\frac{A^*_{B,r}}{\varepsilon A_{B,r}} \geq 1/\varepsilon$ many buckets, of which at least $1/\varepsilon - 1$ contain squares of total area $\varepsilon A_{B,r}$ (the last bucket might not be full and contain less squares). Each square is contained in at most two buckets (it might be split in the creation of buckets, however, we have $\varepsilon A_{B,r} \geq k_r^2$ by our assumption $a_{B,r} \geq 1/\varepsilon$ so it cannot be spread over more than two buckets). Hence, in one of the buckets, (parts of) squares of total profit at most a $O(\varepsilon)$ fraction of the total profit of all squares of I_r packed in B are contained. Discard these squares, and thus the total area of the remaining items is at most $A_{B,r}$ while their profit is at least a $(1-O(\varepsilon))$ fraction of the profit we obtained from I_r squares packed in B before.

For the second case, asumme that $L^* < 1/\varepsilon$, which means that $L^* = 0$. That means, if we define as before $a_{B,r} = L^* = 0$, we will not pack any I_r squares into B, but these squares might carry a significant amount of profit. In this case, we want to

*over*estimate the area instead, i.e., we set $a_{B,r} = \min(L'_r \setminus \{0\}) \leq (1+\varepsilon)/\varepsilon$. However, now we need to argue that items of this area still fit into B, especially as it could happen that many groups I_r have $a^*_{B,r} < 1/\varepsilon$ for one block B (i.e. B contains squares of many different groups, and of each group only few squares are packed in B). Therefore, we want to bound the total area by which we overload block B for all these groups. To formalize this, let S denote the set of all indices r s.t. $k_r \in K_L$ and group r has $L^* < 1/\varepsilon$. For such r, we have $1/\varepsilon \leq a_{B,r} \leq (1+\varepsilon)/\varepsilon$.

So far, we only have the upper bound $(1+\varepsilon)k_r^2/\varepsilon$ on $A_{B,r}$, which might be large compared to the area of B. Here, we use the second property that we have proven in lemma 35: either $k_r \leq \varepsilon^2 \min\{w_B, h_B\}$, or $k_{r-1} \geq \varepsilon^2 \min\{w_B, h_B\}$, which means that if any square of the group I_r is packed into B, then the upper bound for square sizes in this group, k_r, is at most $\varepsilon^2 \min\{w_B, h_B\}$. Thus we get $A_{B,r} \leq (1+\varepsilon)(1/\varepsilon k_r^2) \leq (1+\varepsilon)1/\varepsilon \cdot \varepsilon^4 \min\{w_B, h_B\}^2 = O(\varepsilon^3)a_B$.

Also note that for $r' \in S$ such that $k_r > k_{r'}$,

$$\begin{aligned}
& \frac{k_r}{k_{r'}} > 1+\varepsilon \\
\Rightarrow \quad & \left(\frac{k_r}{k_{r'}}\right)^2 > (1+\varepsilon)^2 \\
\Rightarrow \quad & \frac{\frac{1}{\varepsilon}k_r^2}{\frac{1}{\varepsilon}k_{r'}^2 \cdot (1+\varepsilon)} > 1+\varepsilon \\
\Rightarrow \quad & \frac{A_{B,r}}{A_{B,r'}} > 1+\varepsilon
\end{aligned}$$

where the last step follows from the fact that $\frac{1}{\varepsilon}k_r^2 \leq A_{B,r} \leq \frac{1}{\varepsilon}k_r^2 \cdot (1+\varepsilon)$ (which holds for both r and r', as both are in S).

Sort the indices in S in decreasing order of the size of the corresponding squares; let $r_1, r_2, \ldots$ be this sorted sequence. We know that two of the corresponding k_r-values of S are always at least a factor $1+\varepsilon$ apart (as we are talking about values in K_L). Using a geometric sum argument, we have that $\sum_{r_i \in S} A_{B,r_i} = A_{B,r_1} + A_{B,r_2} + A_{B,r_3} + \ldots \leq A_{B,r_1} + \frac{1}{1+\varepsilon}A_{B,r_1} + \frac{1}{(1+\varepsilon)^2}A_{B,r_1} + \ldots = A_{B,r_1} \sum_{i=0}^{|S|-1} (\frac{1}{1+\varepsilon})^i = A_{B,r_1} \frac{1-(\frac{1}{1+\varepsilon})^{|S|}}{1-\frac{1}{1+\varepsilon}} \leq O(\varepsilon^3)a_B \frac{1+\varepsilon}{\varepsilon} = O(\varepsilon)a_B$, i.e. the total area of all these excess squares is at most an ε fraction of the total area of B. Thus, with similar arguments as before, we can remove squares of total area at least this excess area from B at cost of at most an $O(\varepsilon)$ fraction of the total profit and hence the lemma holds.

Case 2: $k_r \in K_S$ Now, we consider a k_r-value that is in K_S, i.e., we need to guess the number of squares of this group that are packed in B. We define $L''_r = \left\{ \lfloor (1+\varepsilon)^i \rfloor \;\middle|\; 1 \leq (1+\varepsilon)^i \leq n \right\} \cup \{0\}$, and clearly $|L''_r| \leq O(\log_{1+\varepsilon} n)$. Let $n^*_{B,r}$ denote the number of squares of group I_r that are packed in B, and let S be the set of these squares; we want to guess a value $n_{B,r} \leq n^*_{B,r}, n_{B,r} \in L''_r$, such that the $n_{B,r}$ most profitable squares from S still give profit at least a $\frac{1}{1+\varepsilon}$ fraction of the profit of S.

Let $n_{B,r}$ be the largest value in L''_r that is smaller than $n^*_{B,r}$, i.e. $n_{B,r} = (1+\varepsilon)^i$ for some i and $n^*_{B,r} < (1+\varepsilon)^{i+1}$. The difference between the two numbers is $n^*_{B,r} - n_{B,r} < \varepsilon n_{B,r} \leq \varepsilon n^*_{B,r}$ similar as before. Therefore, if we select the $n_{B,r}$ most profitable squares from S, they have profit at least $1-\varepsilon$ times the profit of S. It could now however happen that these squares selected by the algorithm have larger area than the ones

selected by the optimal solution and hence do not fit into the block. However, as these squares belong to groups I_r with $r \in K_S$, we know, that they are by at most a factor $1+\varepsilon$ larger than the squares selected by the optimal solution. Thus the lemma is proven. □

Using lemma 37 we guess the values $a_{B,r}$ and $n_{B,r}$ for each of the $O_\varepsilon(1)$ blocks B and each of the $O_\varepsilon(1)$ values r in time $(\log n)^{O_\varepsilon(1)}$.

7.4.1 Indirect guessing

We perform the indirect guessing of the values $k_1, k_2, \ldots$ similarly as in section 7.3. Recall that $k_0 = 0$. We define a function $\overline{p}_1(x)$ that *in an approximative sense* maps each value x to the maximum profit that we can obtain from the group I_1 if we defined I_1 to contain all squares i with $s_i \in (0, x]$. Formally, we define $\overline{p}_1(x)$ to be the maximum profit of a subset $\tilde{I} \subseteq \{i \mid s_i \in (0,x]\}$ such that we can assign n_1 squares from $\tilde{I}$ to the rows $F_{C,\ell,j}$ with $f(C,\ell,j) = 1$ and the remaining squares of $\tilde{I}$ can be assigned to the blocks such that for each block B either the squares of $\tilde{I}$ assigned to B have a total area of at most $a_{B,1}$ (since $k_1 \in K_L$; later, for $k_r \in K_S$ we replace the latter condition by requiring that at most $n_{B,r}$ squares of the respective set $\tilde{I}$ are assigned to B if $k_r \in K_S$). We say that such a set $\tilde{I}$ *fits into the space for group* I_1. Unfortunately, in contrast to section 7.3 it is NP-hard to compute $\overline{p}_1(x)$ since it is a generalization of the multi-dimensional knapsack problem. However, in polynomial time we can compute an approximation for it, using that $a_{B,r} \in \{0\} \cup [\frac{1}{\varepsilon}k_r, \infty)$ for each B, r with $k_r \in K_L$.

Lemma 38. *In time $O_\varepsilon(1)\cdot n^{O(1)}$ we can compute a function $\tilde{p}_1(x)$ such that $\frac{1}{1+\varepsilon}\overline{p}_1(x) \le \tilde{p}_1(x) \le \overline{p}_1(x)$ for each x. The function $\tilde{p}_1(x)$ is a step-function with $O(n)$ steps. Moreover, for each x in time $O_\varepsilon(1)\cdot n^{O(1)}$ we can compute a set $\tilde{I} \subseteq \{i \mid s_i \in (0,x]\}$ with $p(\tilde{I}) = \tilde{p}_1(x)$ such that $\tilde{I}$ fits into the space for group I_1.*

The proof for this uses LP-rounding similarly as in [68].

Proof. As before, we can argue that $\overline{p}_1(x)$ is a step-function with at most n steps: Let $\overline{s}_1, \ldots, \overline{s}_{n'}$ be the distinct sizes of squares in I_1. The value of the function $\overline{p}_1(x)$ will only change when the set $\overline{I}_1 = \{i | s_i \in (0,x]\}$ changes, which is only the case when x is one of these $\overline{s}_i$-values. For the function $\tilde{p}_1(x)$, we define the values at points $\overline{s}_i$ to be the values of $\overline{p}_1(x)$ at these points; in between two such points, $\tilde{p}_1(x)$ is defined to be constant.

In order to compute $\overline{p}_1$, we therefore need to evaluate for each $\overline{s}_i$, what is the maximum profit we can gain from a set $\tilde{I}_1 \subseteq \overline{I}_1$ such that $\tilde{I}_1$ fits into the space for I_1. Solving this exactly is NP-hard, thus we compute an approximation for this function, $\tilde{p}_1(x)$. We formulate the problem as an LP and then perform a rounding similar to [68] (which is based on a method for the generalized assignment problem [100]). We describe the following process for some fixed value of r. For such a value, we do all that follows for each $\bar{s}_i$ separately; let x denote the currently considered value.

Notation and LP formulation When considering a certain value r, let $\mathcal{R}_r$ be the set of all row subframes of height (for horizontal row subframes) or width (for vertical row subframes) k_r. I_r now denotes all items with size in $(k_{r-1}, x]$ (note that k_{r-1} was guessed before). For a square i, let $\mathcal{B}_i$ be the set of all blocks B such that

$s_i \leq \varepsilon^2 \min\{w_B, h_B\}$, where w_B, h_B are the width and height of B (which we guessed already). We introduce binary variables $x_{i,R}$ for each square $i \in I_r$ and $R \in \mathcal{R}_r$ that indicates whether square i is packed in row subframe R, and variables $x_{i,B}$ for each square $i \in I_r$ and block $B \in \mathcal{B}_i$ (i.e. we only introduce these variables for pairs of squares and blocks s.t. the square is small enough to be packed in the block). Now, consider the following LP, which assumes that $k_r \in K_L$:

$$\begin{aligned}
\max \quad & \textstyle\sum_{i \in I_r} \left(p_i \cdot \left(\sum_{R \in \mathcal{R}_r} x_{i,R} + \sum_{B \in \mathcal{B}_i} x_{i,B} \right) \right) \\
\text{s.t.} \quad & \sum_{i \in I_r} x_{i,R} \leq \overline{w}_R && \forall R \in \mathcal{R}_r && (7.1) \\
& \sum_{i \in I_r} x_{i,B} \cdot s_i^2 \leq a_{B,r} \cdot x^2 && \forall \text{ blocks } B && (7.2) \\
& \sum_{R \in \mathcal{R}_r} x_{i,R} + \sum_{B \in \mathcal{B}_i} x_{i,B} \leq 1 && \forall i \in I_r && (7.3) \\
& x_{i,B} \geq 0 && \forall i \in I_r, \forall B \in \mathcal{B}_i \\
& x_{i,R} \geq 0 && \forall i \in I_r, \forall R \in \mathcal{R}_r
\end{aligned}$$

Constraint (7.1) ensures that the number of squares packed into each row subframe is at most the number of squares that should be packed in this row subframe according to our guess. Constraint (7.2) ensures that the total area of squares from this group packed into any block B does not exceed the guessed area $a_{B,r}$. Constraint (7.3) ensures that each square is packed at most once. If instead $k_r \in K_S$, we replace constraint (7.2) by $\sum_{i \in I_r} x_{i,B} \leq n_{B,r}$. This LP gives an upper bound for OPT, given that our guesses for $\overline{w}_R, a_{B,r}$, and $n_{B,r}$ are correct. Note that the number of variables is at most $n \cdot O(1/\varepsilon^{10} F(\varepsilon)^3)$, and the number of constraints is also at most $n \cdot O(1/\varepsilon^{10} F(\varepsilon)^3)$. Therefore, we can solve this LP in EPTAS-time $n^{O(1)} \cdot O_\varepsilon(1)$.

Rounding the fractional LP solution We can compute a vertex solution to this LP and we will now discuss how to round this solution x^* to a feasible integral one. We create a bipartite graph such that the fractional LP-solution corresponds to a fractional matching in this bipartite graph. The vertices on the left side of the graph correspond to the squares in I_r; call these vertices $v_1, \ldots, v_{|I_r|}$. The vertices on the right side of the graph correspond to the row subframes and blocks in the following way: for each row subframe R of height k_r (or width k_r, if it is a vertical one), we create $\overline{w}_R$ many vertices $v_1^R, \ldots, v_{\overline{w}_R}^R$, for each block B that can get squares from I_r, we create $t_B = \lceil \sum_{i \in I_r} x_{i,B}^* \rceil \leq n$ many vertices $v_1^B, \ldots, v_{t_B}^B$. Now, we will define edges and at the same time specify the fractional matching M^{fr} by defining values $y_e \in (0, 1]$ for each edge e.

Consider the left hand side vertices in some order $v_1, \ldots, v_{|I_r|}$ and consider a single row $R \in \mathcal{R}_r$; for vertex v_i, denote by f_i the remaining fraction of edges, initially $x_{i,R}^*$, and for a vertex v_k^R, denote by f_k^R the sum over y_e for all incident edges e (initially this is zero). Let the current vertex be v_i, let v_k^R be the vertex for R with the smallest index k such that $f_k^R < 1$. Create an edge between v_i and v_k^R, and let $y_{v_i, v_k^R} = \max\{f_i, 1 - f_k^R\} =: p$. Set $f_i := f_i - p$. Repeat this with the next v_{k+1}^R until $f_i = 0$, then go on to the next vertex v_{i+1}. Now, we define the fractional matching

M^{fr} to be the set of all edges, each edge e having weight y_e. Note that the weights y_e assigned to the edges incident to a certain left side vertex v_i sum to $x^*_{i,R}$.

Now, for a block B, we consider the squares fractionally assigned to B by the LP solution in decreasing order of their area. For the B-vertices we do the same procedure as for the R-vertices and consider the squares in this order to create edges and augment our fractional matching (here, initially $f_i = x^*_{i,B}$). Finally, to each edge incident to a left side vertex v_i, we assign profit p_i. This way, the total profit of M^{fr}, $\sum_{e=(v_i,u)\in G} y_e p_i$, is the same as the objective value of the LP for x^*, which is $\sum_{i,R,B}(x^*_{i,R} + x^*_{i,B})p_i$. The number of nodes in this graph is n on the left side and $O_\varepsilon(1)\cdot n$ on the right side.

We can now find an integral matching in this graph with at least the same profit as M^{fr} (this is a standard result in matching theory, see e.g. Lovász and Plummer [89]; the running time can be bounded by $O(|V|^4)$, where $|V|$ is the number of nodes, i.e., running time $O_\varepsilon(1)\cdot n^4$ in our case). It remains to show how to construct a solution for our original packing problem from this integral matching. First of all, if an edge between v_i and v^R_j is taken by the integral matching M^{int}, we assign this square i to row R. By construction of the graph, we cannot assign more than $\overline{w}_R$ squares to a specific row subframe, and these squares surely fit into the row subframe.

Now, consider squares matched to block-vertices. First assume that $k_r \in K_L$ and thus, we used the area-based block constraints in the LP. It might now be the case that the total area assigned to this block by the LP (in a fractional way) is less than the total area of the integral assignment, hence we cannot assign all squares that are matched to this block in the integral solution to this block without violating the constraint. We will now argue that it is nevertheless possible to assign squares to B that fulfill the constraint and have profit at least $\frac{1}{1+\varepsilon}$ of the profit of all squares matched to B in the integral matching.

Consider a specific block B, and let $x_{i,B,k}$ denote the fraction with which the edge (v_i, v^B_k) was selected in M^{fr}. We call the area that is assigned to B in the fractional solution due to vertex v^B_k the *load of this vertex*. This load is defined as $L^B_k := \sum_{i\in I_r} x_{i,B,k} s_i^2$. Now, denote by v_{i_k} the left hand side vertex of the graph that is matched to the right hand side vertex v^B_k in M^{int}. Due to the sorting of the squares in decreasing order of size when creating edges and M^{fr}, and due to the fact that $\sum_i x_{i,B,k} = 1$ for $k < t_B$, we know that the load of vertex v^B_k is at least $s^2_{i_{k+1}}$ for $k < t_B$: all vertices having edges to v^B_k have area at least as large as square i_{k+1}, as this square has an edge to v^B_{k+1}. Now, it follows that the total load of all vertices from the LP solution is $LP_B := \sum_{k=1}^{t_B} L^B_k \geq \sum_{k=1}^{t_B-1} s^2_{i_{k+1}} = \sum_{k=2}^{t_B} s^2_{i_k}$, which is the total area assigned to B by the integral matching for vertices $v^B_2, \ldots, v^B_{t_B}$. Hence, these squares must fit into the area in B reserved for I_r squares, due to constraint (7.2) in the LP. Thus, the integral solution might exceed this area only due to the square i_1, which is matched to vertex v^B_1. Now it comes into play that we guaranteed in lemma 37 that if $a_{B,r} > 0$, then we have $a_{B,r} \geq 1/\varepsilon$. Partition the squares into buckets in the following way: Consider squares in any order, add squares to one bucket until the total area of squares in this bucket is at least k_r^2; then start filling the next bucket. The area of squares within one bucket is at most $2k_r^2$ as each square has area at most k_r^2, thus we have at least $2/\varepsilon$ many buckets. The profit in the least-profitable bucket is hence at most an $O(\varepsilon)$ fraction of the total profit of these squares, and we can remove them. The free area is at least k_r^2, which fits the excess square i_1.

Now, assume that $k_r \in K_S$. In this case, we replace constraint (7.2) by the simpler

$\sum_{i \in I_r} x_{i,B} \leq n_{B,r}$. We need to argue that again for each block B, the squares assigned by the integral solution do not violate this constraint. In this case, $t_B \leq n_{B,r}$, thus we have at most that many squares matched to this block, hence the constraint is trivially satisfied by the integral solution. □

Like before, we guess a value $k_1' \leq k_1$ and for k_1' we allow only those values where $\tilde{p}_1(x)$ increases by a factor of $1+\varepsilon$. Therefore, there are only $O_\varepsilon(\log n)$ possibilities. We define $I_1' := \left\{ i \mid s_i \in (0, k_1'] \right\}$. We iterate as in section 7.3. We next guess a value k_2' and we define the function $\overline{p}_2(x)$ similarly as above. Note that $\overline{p}_2(k_2) \geq p(\mathrm{OPT} \cap I_2)$ since $k_1' \leq k_1$ and our condition for when a set $\tilde{I}$ fits into the space for group I_1 is a relaxation.

Again, we can compute an approximation $\tilde{p}_2(x)$ for $\overline{p}_2(x)$. Overall, there are $(\log n)^{O_\varepsilon(1)}$ guesses and we can enumerate all of them in running time $O_\varepsilon(n) \cdot (\log n)^{O_\varepsilon(1)}$.

Lemma 39. *In time $O_\varepsilon(n) \cdot (\log n)^{O_\varepsilon(1)} = O_\varepsilon(1) \cdot n^{O(1)}$ we can enumerate guesses for all values k_j such that for one such guess $k_1', k_2', \ldots$ we have for each j that $k_j' \leq k_j$ and there is a set $\tilde{I}_j \subseteq I_j' = \left\{ i \mid s_i \in (k_{j-1}', k_j'] \right\}$ with $p(\mathrm{OPT} \cap I_j) \leq (1 + O(\varepsilon)) p(\tilde{I}_j)$.*

Proof. We iterate the process described in lemma 38 for all k_r-values: First, compute the function $\tilde{p}_1(x)$ (i.e., compute for all square sizes s_i the value $\tilde{p}_1(s_i)$) in time $O_\varepsilon(1) \cdot n^{O(1)}$ by solving the LP and then finding an integral matching in the bipartite graph defined by this LP solution as described in the proof of lemma 38). We get, as before, $O(\frac{\log n}{\log(1+\varepsilon)})$ many candidate values for k_1 at those points where the function "jumps" over a power of $1+\varepsilon$, and guess the correct one, k_1'. So far, this takes time $O_\varepsilon(1) \cdot n^{O(1)} + O(\frac{\log n}{\log(1+\varepsilon)})$. For each of the k_1'-guesses, we now compute in the same manner the function $\tilde{p}_2(x)$, which gives $O(\frac{\log n}{\log(1+\varepsilon)})$ possible choices for k_2'. Note here that $\tilde{p}_2(k_2') \geq p_2(k_2)$, as $k_1' \leq k_1, k_2' \leq k_2$, and again the corresponding LP gives an upper bound on the optimal solution. We continue like that until we have guessed all k_r'-values. The total running time becomes $O_\varepsilon(1) \cdot n^{O(1)} + O(\frac{\log n}{\log(1+\varepsilon)}) \cdot (O_\varepsilon(1) \cdot n^{O(1)} + O(\frac{\log n}{\log(1+\varepsilon)}) \cdot (\ldots)) = \left(\frac{\log n}{\log(1+\varepsilon)}\right)^{O(1/\varepsilon^{10} F(\varepsilon)^3)} \cdot O_\varepsilon(1) \cdot n^{O(1)} = O_\varepsilon(1) \cdot n^{O(1)}$. □

7.5 An EPTAS for geometric knapsack with squares

The guessed values k_j' imply the heights and widths of the elongated cells in the following way. Consider an elongated cell C and assume w.l.o.g. that $h_C \leq w_C$. Then h_C is the maximum height of a frame of C, i.e., $h_C := \max_\ell \sum_j h_{C,\ell,j}$ where for each row subframe $F_{C,\ell,j}$ we have that $h_{C,\ell,j} := k'_{f(C,\ell,j)}$ (for block subframes we already guessed the height and the width). The width w_C of C is the total width of all its frames, i.e., $w_C := \sum_\ell w_{C,\ell}$ where for each frame $F_{C,\ell}$ its width is the maximum width of a subframe, i.e., $w_{C,\ell} := \max_j w_{C,\ell,j}$ and for each row subframe $F_{C,\ell,j}$ we define $w_{C,\ell,j} := \overline{w}_{C,\ell,j} \cdot k_{f(C,\ell,j)}$ (again, for block subframes we already guessed the height and the width).

We obtained $O(F(\varepsilon))$ large squares and $O(F(\varepsilon))$ cells for which we now know all heights and widths. We verify in $O_\varepsilon(1)$ time that there exists a feasible packing for them (if not then we reject our guess). Observe that we *under*estimated all guessed quantities and therefore for the correct guesses a feasible packing must exist.

For each group I'_j we obtained a set $\tilde{I}_j$ that fits into the space of group I_j. We take the union of all these sets $\tilde{I}_j$. This is not yet a feasible solution: even though each block B gets squares assigned whose total size does not exceed the size of B, a feasible packing is not guaranteed to exist. However, we can remove some of the squares such that we lose at most a factor of $1+\varepsilon$ and the squares assigned to the blocks can be packed in a greedy manner. Thus, we obtain a globally feasible solution $\tilde{I}$ with $p(\tilde{I}) \geq (1+O(\varepsilon))^{-1} \sum_j p(\tilde{I}_j) \geq (1+O(\varepsilon))^{-1}\mathrm{OPT}$.

Theorem 9. *There is a $(1+\varepsilon)$-approximation algorithm for the two-dimensional knapsack problem for squares with a running time of $2^{2^{O(1/\varepsilon^4)}} \cdot n^{O(1)} = O_\varepsilon(1) \cdot n^{O(1)}$.*

Proof. We will now discuss step by step how our algorithm proceeds and which running time each step incurs.

Step 1: Guessing large squares and number of cells, frames and subframes As described in section 7.2, we guess B in time $O(F(\varepsilon))$, then we guess the at most B large squares in time $\left(\frac{1}{\log(1+\varepsilon)}\right)^{O(F(\varepsilon))} \cdot n$.

Guessing the number of cells (block cells and elongated cells) takes at most $O(1/\varepsilon^{10}F(\varepsilon)^3)$ time, guessing the number of frames for each elongated cell takes total time $(1/\varepsilon)^{O(1/\varepsilon^{10}F(\varepsilon)^3)}$, and guessing the number of subframes for each frame takes total time $(1/\varepsilon)^{O(1/\varepsilon^{14}F(\varepsilon)^3)}$. This gives a total running time of $\left(\frac{1}{\varepsilon\log(1+\varepsilon)}\right)^{O(1/\varepsilon^{14}F(\varepsilon)^3)} \cdot n$ for this step.

Step 2: Guessing dimensions of blocks As described in section 7.3, we first guess $\tilde{\alpha}$ in time $\frac{n\log n}{\log(1+\varepsilon)}$. Guessing the height and width of one block then takes time $O(\frac{\log n}{\log(1+\varepsilon)})$. The number of blocks in the whole knapsack is the initial number of elongated cells in the knapsack (i.e. before application of lemma 35, as these do not create new block subframes although they increase the number of elongated cells) times $O(1/\varepsilon^2)$ (as in each frame we might have at most one block subframe) plus the number of block cells, i.e., in total we have at most $O(1/\varepsilon^2 F(\varepsilon) + F(\varepsilon)) = O(1/\varepsilon^2 F(\varepsilon))$ blocks. Therefore, the total time for step 2 is $\frac{n\log n}{\log(1+\varepsilon)} \cdot (\frac{\log n}{\log(1+\varepsilon)})^{O(1/\varepsilon^2 F(\varepsilon))} = n \cdot (\frac{\log n}{\log(1+\varepsilon)})^{O(1/\varepsilon^2 F(\varepsilon))} = n^2 \cdot (\frac{1}{\log(1+\varepsilon)})^{O(1/\varepsilon^2 F(\varepsilon))}$.

Step 3: Guessing $\overline{w}_{C,\ell,j}$ and $f(C,\ell,j)$ for row subframes After lemma 35, the number of row subframes is at most $O(1/\varepsilon^{10}F(\varepsilon)^3)$. For $\overline{w}_{C,\ell,j}$ there are $\log_{1+\varepsilon} n$ many possibilities, for $f(C,\ell,j)$ there are $O(1/\varepsilon^4 F(\varepsilon)^2)$ many possibilities as described in the proof of lemma 36, and hence we need time

$$\left(\frac{\log n}{\log(1+\varepsilon)} O(1/\varepsilon^4 F(\varepsilon)^2)\right)^{O(1/\varepsilon^{10}F(\varepsilon)^3)} = n \cdot (\frac{1/\varepsilon F(\varepsilon)}{\log(1+\varepsilon)})^{O(1/\varepsilon^{10}F(\varepsilon)^3)}$$

for this step.

Step 4: Guessing area of square groups and number of squares in blocks In the next step, we want to guess the quantities $a_{B,r}$ and $n_{B,r}$ as described in lemma 37 for all blocks B and all groups I_r (number of blocks is at most $O(1/\varepsilon^2 F(\varepsilon))$ and number of groups is at most $O(1/\varepsilon^4 F(\varepsilon)^2)$). For each block B and group I_r, we have $O(\frac{\log n}{\log(1+\varepsilon)})$ many possibilities, hence the total time for step 4 is $(\frac{\log n}{\log(1+\varepsilon)})^{O(1/\varepsilon^6 F(\varepsilon)^3)} = n \cdot (\frac{1}{\log(1+\varepsilon)})^{O(1/\varepsilon^6 F(\varepsilon)^3)}$.

Step 5: Guessing the values k_r There are $O(1/\varepsilon^4 F(\varepsilon)^2)$ values to guess. In order to guess each value, we first need to compute the function $\overline{p}_r(x)$, or, to be more precise, for each square size we need to evaluate the function at this point. Evaluating the function at one point takes time $(n \cdot 1/\varepsilon \cdot F(\varepsilon))^{O(1)}$ for solving the LP and time $(n \cdot 1/\varepsilon \cdot F(\varepsilon))^{O(1)}$ for finding the integral matching (as the graph has at most $O(1/\varepsilon^{10} F(\varepsilon)^3 \cdot n)$ many nodes). Thus guessing all k_r-values can be done in time $n^{O(1)} \cdot O(F(\varepsilon)/\varepsilon)^{O(1)} \cdot (\frac{\log n}{\log(1+\varepsilon)})^{O(1/\varepsilon^4 F(\varepsilon)^2)} = n^{O(1)} \cdot (\frac{F(\varepsilon)}{\varepsilon \log(1+\varepsilon)})^{O(1/\varepsilon^4 F(\varepsilon)^3)}$.

Step 6: Checking whether cells and large squares can be packed into the knapsack We need to enumerate all possible packings of these rectangular regions into the knapsack. When packing L rectangles into one rectangle, we need running time $L^{O(L)}$, hence in this case we obtain a total running time of $(F(\varepsilon) + 1/\varepsilon^{10} F(\varepsilon)^3)^{O(F(\varepsilon)+1/\varepsilon^{10} F(\varepsilon)^3)} = (F(\varepsilon)/\varepsilon)^{O(1/\varepsilon^{10} F(\varepsilon)^3)}$.

Step 7: Packing squares into blocks and row subframes Notice that, in contrast to lemma 30, we do not know beforehand which squares can go into which blocks, as some squares might be small compared to one block (i.e., the square's size is at most an ε^2 fraction of the block's side lengths) but not compared to another block. However, a feasible assignment of squares to blocks is given by the integral matching computed as described in the proof of lemma 38, as well as an assignment of squares to the row subframes. We simply pack squares into the corresponding row subframes, and squares assigned to blocks are packed using NFDH (running time $n^{O(1)}$). We know from the matching solution and lemma 37 that the area of the selected squares exceeds the area of the block by at most a factor $1 + \varepsilon$. We can apply lemma 26 to ensure that we pack a subset of these squares of profit at least $1 - O(\varepsilon)$ times the total profit of assigned squares into each block. □

7.6 Discussion and directions for future work

For the geometric knapsack problem with squares, we presented an EPTAS that improved the running time of the previous known PTAS from $\Omega\left(n^{2^{2^{1/\varepsilon}}}\right)$ to $O_\varepsilon(1) \cdot n^{O(1)}$. In our work, we tried to improve the large running times of some of the PTAS-steps. Our result benefits from not guessing certain values and quantities directly, but rather analysing carefully how the profit changes with these values, reducing the number of candidate values. Thus we speed up the guessing process significantly. In essence, we use structural information about the input as follows:

- When guessing the large items, we make use of the fact that among a group of items with similar profit, it is always the best option to choose the smallest ones as large items, instead of testing every possible choice.

- When guessing the structure of elongated cells, recall that we need to find some items in the input whose sizes give us the heights of the row subframes. We showed that when a factor of $1 + \varepsilon$ does not matter, only few input items make a difference for the overall profit of the solution, thus allowing us to reduce the number of candidate values significantly.

The larger running time of the PTAS resulted in parts from standard techniques such as the shifting technique and more fundamentally from guessing certain quantities out of large candidate sets. Such approaches are widely applied to geometric problems and often yield these large running times. Our result shows that such running times are not always necessary and can be overcome by guessing certain items and parameters more carefully. It would be interesting to see if our ideas can be transferred to other problems that use similar techniques.

8 Geometric Knapsack with Resource Augmentation

In this chapter, we study the two-dimensional geometric knapsack problem with a square knapsack and rectangular items, a more general problem setting than the one considered in the previous chapter. However, we allow ourselves two-dimensional resource augmentation in this setting, meaning that, while the optimal solution uses a knapsack of size $N \times N$, the algorithm is allowed to use a knapsack of size $(1+\varepsilon)N \times (1+\varepsilon)N$. For this problem, the best known PTAS so far was given by Fishkin et al. [44]. It has a running time of the form $\Omega\left(n^{1/\varepsilon^{1/\varepsilon}}\right)$, i.e., the exponent of n is double exponential in $1/\varepsilon$.

Results in this chapter and organization In this chapter, we give an algorithm with running time of the form $O_\varepsilon(1) \cdot n^{O(1)}$ and which even computes an *optimal solution*. The improvement in the running time results from new guessing techniques for the selection and placement of the large squares, while the improvement in approximation factor (from $1+\varepsilon$ to 1) is made possible by using the resource augmentation. In section 8.1, we describe the classification of the rectangles into different sets. In section 8.2, we describe a grid inside the knapsack that gives the desired structure of the packing. Finally, in section 8.3 we describe how to pack items into this structure. We first focus on the case where rotation of the items is not allowed; at the end of section 8.3 we talk shortly about the setting with rotation.

8.1 Rectangle classification

We classify rectangles into large, medium, horizontal, vertical, and small rectangles according to their side lengths, using constants $\mu, \delta > 0$ defined below. We use a separate routine for the medium rectangles. Using the next lemma we ensure that they have small total area in the optimal solution.

Lemma 40. *There is a universal set $D = \{(\mu_0, \delta_0), (\mu_1, \delta_1), \ldots, (\mu_{1/\varepsilon}, \delta_{1/\varepsilon})\}$ with $\mu_i = \varepsilon^2 \cdot \delta_i^3 < \varepsilon$ and $\mu_i, \delta_i \in \Omega_\varepsilon(1)$ for each $(\mu_i, \delta_i) \in D$ such that for each input there exists a pair $(\mu, \delta) \in D$ and the total area of rectangles $i \in \mathrm{OPT}$ with $\mu \cdot N < w_i < \delta \cdot N$ or $\mu \cdot N < h_i < \delta \cdot N$ is bounded by $2\varepsilon \cdot N^2$.*

Proof. Our construction is similar to [44]. Define $\delta_j := \varepsilon^{2 \cdot 3^j - 1}$ and $\mu_j := \delta_{j+1} = \varepsilon^{2 \cdot 3^{j+1} - 1}$ for $0 \le j \le 1/\varepsilon$. Define $M_j^V := \left\{ i \in \mathrm{OPT} \mid \mu_j N < h_i < \delta_j N \right\}$ and $M_j^H := \left\{ i \in \mathrm{OPT} \mid \mu_j N < w_i < \delta N \right\}$. It holds that $M_j^V \cap M_k^V = \emptyset$ and $M_j^H \cap M_k^H = \emptyset$ for $j \neq k$, as the intervals (μ_j, δ_j) are disjoint. Therefore, each rectangle from the optimal solution occurs in at most one of the sets M_j^H and at most one of the sets M_j^V, and thus $\sum_{j=1}^{1/\varepsilon} a(M_j^H \cup M_j^V) \le 2N^2$ as the total area of all rectangles in OPT is at most

N^2 (we denote by $a(S)$ the total area of the rectangles in set S). Therefore, there is one value j such that $a(M_j^H \cup M_j^V) \leq 2\varepsilon N^2$. □

We guess the correct pair $(\mu, \delta) \in D$ due to lemma 40. Note that μ might be as small as $\varepsilon^{O(1/\varepsilon)}$. We denote by L the set of all rectangles i with $w_i > \delta \cdot N$ and $h_i > \delta \cdot N$, by V all rectangles i with $w_i < \mu \cdot N$ and $h_i > \delta \cdot N$, by H all rectangles i with $w_i > \delta \cdot N$ and $h_i < \mu \cdot N$, and by S all rectangles i with $w_i < \mu \cdot N$ and $h_i < \mu \cdot N$. We define $M := I \setminus (L \cup V \cup H \cup S)$, i.e., all rectangles i with $\mu \cdot N < w_i < \delta \cdot N$ or $\mu \cdot N < h_i < \delta \cdot N$.

We treat the rectangles in M separately using the following lemma and pack them into the additional space gained by increasing the size of the knapsack (a similar argumentation was used in [1]). The intuition is that increasing the size of the knapsack by a factor $1 + O(\varepsilon)$ gains free space that is by a constant factor larger than the total space needed for the medium rectangles. This makes the packing easy.

Lemma 41. *There is a polynomial time algorithm that computes a set $M' \subseteq M$ with $p(M') \geq p(M \cap \mathrm{OPT})$ and a packing for M' into two boxes of sizes $O(\varepsilon) \cdot N \times N$ and $N \times O(\varepsilon) \cdot N$, respectively.*

Proof. We know from lemma 40 that $a(M \cap \mathrm{OPT}) \leq 2\varepsilon N^2$ and for all rectangles in M we have $\mu N < w_i < \delta N$ or $\mu N < h_i < \delta N$. Let $M_V := \{i \in M | \mu N < w_i < \delta N\}$ and $M_H := \{i \in M | \mu N < h_i < \delta N\}$. We want to select subsets $M'_V \subseteq M_V$ and $M'_H \subseteq M_H$ s.t. $a(M'_V) \leq 4\varepsilon N^2$ and $a(M'_H) \leq 4\varepsilon N^2$. Then, we can use the algorithm NFDH from [28], which packs M'_H into an area of width N and height $2a(M'_H)/N + h_{max}$ where h_{max} denotes the maximum height of rectangles in M'_H. As rectangles in M'_H have height at most $\delta N \leq \varepsilon N$, we need height at most $2 \cdot 4 \cdot \varepsilon N^2/N + \varepsilon N = O(\varepsilon)N$. Analogously we can pack the rectangles M'_V into an area of height N and width $O(\varepsilon)N$.

For selecting the rectangles M'_H, we consider a knapsack problem: For each rectangle $i \in M_H$, we create an item with size equal to the area of i and profit equal to the profit of i. The knapsack has capacity $2\varepsilon N^2$. We use the simple greedy Knapsack algorithm with one modification: the last item (that is not taken into the Knapsack because it would exceed the capacity) is also selected. Since this item has size at most $2\varepsilon N^2$, we get that all selected items have total size at most $4\varepsilon N^2$ (and thus all rectangles from M_H corresponding to these items have total area at most $4\varepsilon N^2$). Moreover, the selected rectangles have profit at least $p(M_H \cap \mathrm{OPT})$. The same selection procedure can be applied to rectangles in M_V. □

Lemma 41 ensures that $p(M') \geq p(M \cap \mathrm{OPT})$ and thus our medium rectangles are as profitable as the medium rectangles in the optimal solution. In contrast, the algorithm in [44] loses a factor of $1 + \varepsilon$ in the approximation ratio in this step. From now on we consider only the rectangles in $L \cup H \cup V \cup S$.

8.2 Placing a grid

Using the argumentation in [44], by enlarging the knapsack by a factor $1 + \varepsilon$ we can round up the heights of the rectangles in $L \cup V$ and the widths of the rectangles in $L \cup H$ to integral multiples of $\varepsilon \cdot \delta \cdot N$. Even more, we can ensure that they are aligned with a grid $\mathcal{G}$ of granularity $\varepsilon \cdot \delta \cdot N$, see fig. 8.1. Formally, we define $\mathcal{G} := \{(x, k \cdot \varepsilon\delta N) \mid k \in \mathbb{N}, x \in \mathbb{R}\} \cup \{(k \cdot \varepsilon\delta N, x) \mid k \in \mathbb{N}, x \in \mathbb{R}\}$.

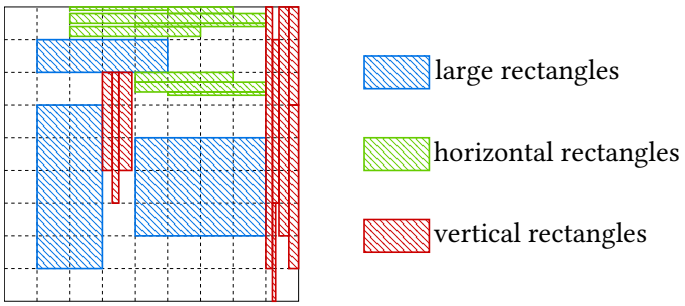


Figure 8.1: The grid $\mathcal{G}$ and rectangles aligned with it.

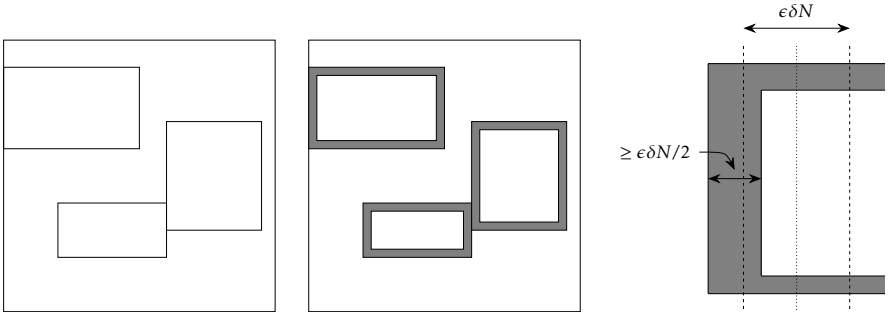
Lemma 42 ([44]). *By enlarging the knapsack by a factor $1+\varepsilon$ we can assume for the input and the optimal solution that*

- *for each $i \in L \cup H$ we have that w_i and the x-coordinates of all corners of i are integral multiples of $\varepsilon\delta N$,*
- *for each $i \in L \cup V$ we have that h_i and the y-coordinates of all corners of i are integral multiples of $\varepsilon\delta N$,*
- *the height and width of the knapsack is an integral multiple of $\varepsilon\delta N$.*

Proof. We follow the argumentation from [44]. Enlarge the knapsack and rectangles in $L \cup H \cup V$ by a factor of $1+\varepsilon$ (the packing is still feasible). Define the *induced space* of some rectangle $i \in L \cup H \cup V$ to be the space that i occupies in this enlarged packing. Reduce the rectangles (but not the knapsack) back to their original size. Consider some rectangle i in $L \cup H$; we will shift i horizontally inside its induced space in order to align it with the grid (the argumentation is analogous for rectangles in $L \cup V$ to align them vertically with the grid). For illustration see fig. 8.2b. The distance between a vertical boundary of i and the corresponding vertical boundary of its induced space is at least $\varepsilon\delta N/2$ as the rectangle has width at least δN. As the grid lines are $\varepsilon\delta N$ apart and we can shift i by $\varepsilon\delta N/2$ to the left or to the right, we can shift it in either direction s.t. one of its vertical boundaries is aligned with one of the grid lines (see fig. 8.2c). Now, we can enlarge the rectangle again by a factor of at most $1+\varepsilon$ until its other vertical boundary is aligned with another grid line. In order to do this (without knowing the optimal solution), we simply round all side lengths to the next larger multiple of $\varepsilon\delta N$. The enlarged rectangle will still reside inside its induced space. Finally, by enlarging the knapsack by a factor $1+\varepsilon$ we can easily ensure that its height and width are integral multiples of $\varepsilon\delta N$. □

8.3 Packing the rectangles

The grid $\mathcal{G}$ divides the knapsack into $1/\delta^2 \leq \left(\frac{1}{\varepsilon}\right)^{O(1/\varepsilon)} = O_\varepsilon(1)$ grid cells. In the packing due to lemma 42, each such cell is either fully covered by a large rectangle or does not intersect a large rectangle at all. At this point, the algorithm in [44] guesses the large rectangles directly which leads to a running time of $\binom{n}{1/\delta^2}$ which might be as large $\Omega\left(n^{1/\varepsilon^{1/\varepsilon}}\right)$ since δ can be as small as $\varepsilon^{1/\varepsilon}$. Instead, we now guess the boundaries of the large rectangles in this packing, without guessing the rectangles themselves.



(a) Initial instance.

(b) Instance after enlarging and then shrinking the rectangles and enlarging the knapsack. Gray areas denote the induced spaces.

(c) Shifting of a large or horizontal rectangle. The dashed lines denote grid lines.

Figure 8.2: For aligning large, horizontal and vertical rectangles with the grid, we first enlarge them, and then shrink them again. The third picture illustrates that we have enough space inside the induced space to align the rectangle with a grid line.

Since there are $O_\varepsilon(1)$ grid cells, this can be done in time $O_\varepsilon(1)$. Given the boundaries, we compute the best choice for the large rectangles using a greedy algorithm.

Lemma 43. *Given the boundaries of the large rectangles in the optimal solution, we can compute a set $L' \subseteq L$ with $p(L') \geq p(L \cap \mathrm{OPT})$ in time $O_\varepsilon(n)$ that fits into the space given by these boundaries.*

Proof. We consider the boundaries one by one in arbitrary order. For each boundary, search for the most profitable rectangle in L that fits into this space, pack it there, and remove it from the list of candidates for the next boundaries. Clearly, the optimal solution cannot pack any more profitable rectangles into these boundaries. □

Each remaining grid cell (i.e., that is not covered by a large rectangle) is either intersected by a horizontal rectangle, or by a vertical rectangle, or by no rectangle in $H \cup V$ at all. First, we guess the total area of the selected horizontal and vertical rectangles in the optimal solution. Then, as in [31, 44], we use techniques from strip packing to compute a profitable packing for the horizontal and vertical rectangles while enlarging the knapsack slightly.

Lemma 44. *When enlarging the knapsack by a factor $1 + O(\varepsilon)$ there is an algorithm with running time $O_\varepsilon(1) \cdot n^{O(1)}$ that computes sets $H' \subseteq H$ and $V' \subseteq V$ and a packing for them in the space of the knapsack that is not occupied by rectangles in L' such that $p(H') \geq p(H \cap \mathrm{OPT})$, $p(V') \geq p(V \cap \mathrm{OPT})$, $a(H' \cup V') \leq a((H \cup V) \cap \mathrm{OPT}) + O(\varepsilon) \cdot N$ and the space of the knapsack not used by rectangles in $L' \cup V' \cup H'$ can be partitioned into $O\left(\left(\frac{1}{\varepsilon\delta}\right)^2\right) = O_\varepsilon(1)$ rectangular boxes.*

Proof. Our reasoning essentially reiterates the argumentation in [31, 44], we refer to these papers for more details. Consider the horizontal rectangles H. They have up to $\frac{1}{\varepsilon\delta}$ many widths and their minimum width is $\varepsilon\delta$. For each width class we guess the total height of the rectangles in OPT of this width in integral multiples of

$\varepsilon^2\delta N$, i.e., for each width class we guess an integer $k \in O(\frac{1}{\varepsilon^3\delta^2})$ such that the guessed height equals $k \cdot \varepsilon^2\delta N$. By increasing the size of the knapsack by a factor $1+\varepsilon$ this discretization is justified since then we gain additional space of width N and height εN which is enough to accommodate one rectangle of height $\varepsilon^2\delta N$ of each width class. Also, we can restrict to values of k with $k \in O(\frac{1}{\varepsilon^3\delta^2})$ since for larger values the total area of the rectangles in this width class would be larger than the size of the knapsack. For each i we denote by A_i the guessed total height of the rectangles of width $i \cdot \varepsilon\delta$. We do a similar operation for the rectangles in V. Note that for the guesses there are only $O_\varepsilon(1)$ possibilities.

Then we guess which of the $O(\varepsilon^2\delta^2)$ grid cells are used by horizontal rectangles and which are used by vertical rectangles. Note that a grid cell can be used by only one of these two types of rectangles so the guessing can be done in time $2^{O(\varepsilon^2\delta^2)} = O_\varepsilon(1)$. Consider the cells that we guessed to be used by horizontal rectangles. For each grid row we partition them into sets of connected components. We call these connected components *blocks*. Then we use a configuration-LP to pack horizontal rectangles into the blocks. Each configuration is a vector $(b_1, \ldots, b_{1/(\varepsilon\delta)})$ with $\sum_i b_i \cdot i \leq 1/(\varepsilon\delta)$ that specifies b_i slots for rectangles of width $b_i \cdot i$ for each i. For each block there are $(\frac{1}{\varepsilon\delta})^{\frac{1}{\varepsilon\delta}}$ many configurations with at most $1/(\varepsilon\delta)$ slots each. Based on this we formulate a configuration LP that ensures that

- in each block the total height of assigned configurations is at most the height of the block, i.e., $\varepsilon\delta N$, and
- in all horizontal blocks B for each rectangle width $i \cdot \varepsilon\delta$ there are slots whose total height is at least A_i.

The resulting LP has $O_\varepsilon(1)$ variables and constraints and we compute an extreme point solution for it. The number of constraints in the LP is bounded by the number of blocks plus the number of rectangle widths, i.e., by $2\left(\frac{1}{\varepsilon\delta}\right)^2$, and thus the number of non-zero entries in our solution is bounded by the same value. This yields a partition of the blocks for horizontal rectangles into at most $2\left(\frac{1}{\varepsilon\delta}\right)^3$ slots, each slot corresponding to one rectangle width. We now fill the rectangles in H fractionally into these slots. We do this in a greedy manner, i.e., for each rectangle width we order the rectangles non-increasingly by density p_i/h_i and assign them greedily in this order into the slots (the slots are ordered arbitrarily). If a rectangle does not fit entirely into the remaining space in a considered slot, we split the rectangle horizontally and assign one part of it to the current slot and the remainder to the next slot (we iterate this procedure if the remaining part does not fit into the next slot). The obtained profit of the rectangles assigned in this way is at least $p(H \cap \mathrm{OPT})$ (also counting the whole profit of the last rectangle that might be only partially assigned). At the end, there are at most $2\left(\frac{1}{\varepsilon\delta}\right)^3$ rectangles that are split. Their total height is bounded by $\mu N \cdot 2\left(\frac{1}{\varepsilon\delta}\right)^3 \leq O(\varepsilon) \cdot N$ and thus we can place them into an additional block of height $O(\varepsilon) \cdot N$ and width N. By enlarging the size of the knapsack by a factor $1+\varepsilon$ we gain additional free space of this size. By construction, the empty space in the so far considered blocks can be partitioned into at most $O\left(\left(\frac{1}{\varepsilon\delta}\right)^2\right)$ rectangular boxes. We do the same operation for the vertical rectangles in V. □

Finally, we add the small rectangles. We select a set of small rectangles whose total area is at most the area of the remaining space which yields an instance of

one-dimensional knapsack. Using that the rectangles are all very small and that we can increase the capacity of our knapsack, we can select small rectangles that are as profitable as the small rectangles in OPT and we can find a packing for all the selected rectangles. In contrast, in this step the algorithm in [44] loses a factor $1+\varepsilon$ in the approximation ratio (instead of increasing the size of the knapsack).

Lemma 45. *When enlarging the knapsack by a factor $1+\varepsilon$ there is a polynomial time algorithm that computes a set of rectangles $S' \subseteq S$ with $p(S') \geq p(S \cap \mathrm{OPT})$ and a packing for S' into the remaining space that is not used by the rectangles in $L' \cup H' \cup V'$.*

Proof. Consider the free space in the knapsack that is not used by $L' \cup H' \cup V'$. According to lemma 44, this space can be partitioned into $O\left(\left(\frac{1}{\varepsilon\delta}\right)^2\right)$ rectangular areas; from now on we will call them *boxes*. Our goal is to find a set of rectangles $S' \subseteq S$ which we can pack into the free space and the area that we gain by resource augmentation, and we want to have $p(S') \geq p(S \cap \mathrm{OPT})$.

To this end we define an instance of the one-dimensional knapsack problem: For each rectangle $i \in S$, create an item i' for the knapsack instance with profit equal to the profit of i and size equal to the area of i. The capacity of the knapsack is the total area of all boxes; denote this quantity by A. For solving this problem we use the simple greedy algorithm for knapsack. This algorithm sorts the items in non-increasing order of their profit-to-size-ratios, and then greedily packs items into the knapsack in this order until an item does not fit. We put all rectangles corresponding to items selected by this algorithm into S' and we also add into S' the rectangle corresponding to the first item that is not taken into the knapsack by the greedy algorithm because it would exceed the capacity A. It has size at most $\mu^2 N^2$. The profit of S' is now at least that of the optimal solution (which is only allowed to use capacity A). This is the case since our profit is at least the profit of the optimal solution to the canonical LP-relaxation for our knapsack instance.

From now on, we use a standard argumentation to place the items in S' greedily into the remaining space (see, e.g., [94]): We ignore all boxes that are too small, i.e. boxes that have one side length $\leq \mu N$. Let k be the number of boxes that are large enough and $\overline{k}$ be the number of boxes that are too small, $k, \overline{k} \leq O\left(\left(\frac{1}{\varepsilon\delta}\right)^2\right)$. The area of a box that is too small is at most μN^2 (its larger side might be as large as N), and thus we lose in total an area of at most $\overline{k}\mu N^2$ by this.

We fill the rectangles S' into all other boxes using NFDH. The total area that is available for this is at least $A - \overline{k}\mu N^2$. Let $B_1, \ldots, B_k$ be the boxes in the order they are used by NFDH. Let $\overline{w}_j, \overline{h}_j$ be the side lengths of B_j and let i_j be the first rectangle packed into B_j (by our constraint on the size of the boxes used we know that at least one rectangle fits into each box). If all rectangles were packed into the boxes, we are done, so assume this is not the case, i.e. some rectangles remain unpacked. For box B_j, the NFDH-algorithm packs rectangles into a sub-box $\hat{B}_j$ of size $\overline{w}_j \times \hat{h}_j$, i.e. a sub-box of height $\overline{h}_j - \hat{h}_j$ and width $\overline{w}_j$ is unused by the algorithm (see fig. 8.3 for illustration). This means that $\overline{h}_j - \hat{h}_j \leq \mu N$ (otherwise it would be possible to pack another rectangle there). In [28] it is proven that the wasted space inside $\hat{B}_j$ is at most $h_{i_j} \cdot \overline{w}_j$. Thus, the total unused space inside B_j is at most $h_{i_j} \cdot \overline{w}_j + \mu N \cdot \overline{w}_j \leq 2\mu N^2$ and the unused space in all boxes is at most $2\mu N^2 \cdot k$. The total unused space in the whole knapsack (i.e. the space not used within $B_1, \ldots, B_k$ and the space in the small

boxes) is therefore at most $2k\mu N^2 + \overline{k}\mu N^2 = O\left(\frac{1}{\varepsilon^2\delta^2}\right)\mu N^2 = O(\varepsilon)N^2$. In addition, we need additional space because the rectangles in S' have area larger than A, but this additional space is again in $O(\varepsilon)N^2$. By again using NFDH and the bounds derived in [28], we can pack all remaining rectangles into an area of width N and height $\mu N + 2 \cdot O(\varepsilon)N^2/N = O(\varepsilon)N$ gained by enlarging the size of the knapsack by a factor $1+\varepsilon$. □

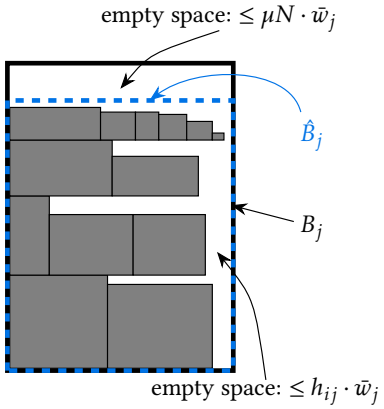


Figure 8.3: Packing produced by NFDH inside one box.

Overall, we obtain a solution whose profit is at least $p(\mathrm{OPT})$ and which is packed into a knapsack of size $(1+O(\varepsilon))N \times (1+O(\varepsilon))N$. If the algorithm is allowed to rotate rectangles by 90 degrees then it is straight forward to adjust lemmas 43 and 44 accordingly. Also, lemma 45 still holds, even if we do not rotate any rectangle in S (while OPT might still do this).

Theorem 10. *For any $\varepsilon > 0$ there is an algorithm with a running time of $O_\varepsilon(1) \cdot n^{O(1)}$ for the two-dimensional geometric knapsack problem for rectangles under $(1+\varepsilon)$-resource augmentation in both dimensions that computes a solution whose profit is at least $p(\mathrm{OPT})$. This holds for the settings with and without rotation.*

8.4 Discussion and directions for future work

In this chapter, we gave an efficient algorithm for the geometric knapsack problem with rectangular items that achieves an optimal solution under resource augmentation. For this result, we again guess some items more carefully in order to reduce the running time. Furthermore, we make use of resource augmentation to improve the approximation factor up to the point where we obtain an optimal solution. This shows that relaxing constraints and assumptions such as the size of the knapsack can be helpful in order to achieve better results than those that are possible without such relaxations. The crucial insight for our result was that the shifting technique incurs a loss of factor $1+\varepsilon$ in the approximation guarantee in order to remove some items from the input. However, these items can easily be taken care of as soon as we augment the size of the knapsack. Such a removal of items in order to allow for an easier packing of the rest of the items is a standard technique in geometric packing problems. Future work could thus try to obtain improved results for other problems

where resource augmentation or some other form of constraint relaxation might be useful to avoid excluding items from the solution and thus to find an optimal solution.

Note that if we assume $\mathsf{P} \neq \mathsf{NP}$, it is not possible to obtain the same result we showed in this chapter using only one-dimensional resource augmentation if the direction of this resource augmentation is fixed. If we could do this, we could also solve the NP-complete partition problem[1] in polynomial time: Assume that the knapsack can only be stretched vertically, then for each number a_i in the partition problem, create an item of width a_i and height $N/2$, and consider a knapsack of size $N \times N$ with $N = \frac{\sum_i a_i}{2}$. Resource augmentation does not help to solve the problem because even if we increase the height of the knapsack slightly, we can still only fit two items on top of each other. Therefore, finding an optimal solution for the knapsack problem in polynomial time would imply finding an optimal solution for the partition problem in polynomial time, contradicting the $\mathsf{P} \neq \mathsf{NP}$ assumption.

It would, however, be interesting to consider the case that the algorithm is allowed to choose the dimension which can be extended.

[1] In the partition problem, we are given n numbers $a_1, \ldots, a_n$ and want to find a partition of these numbers into two sets A_1, A_2 such that the sum of the numbers in A_1 equals the sum of the numbers in A_2.

9 Conclusion

In this thesis, we contributed new algorithms and lower bounds to two fundamental packing problems, online bin packing and the geometric knapsack problem.

For online bin packing, we defined a new algorithm framework called Extreme Harmonic and showed that an algorithm in this framework can yield a competitive ratio of at most 1.5813. Additionally, with a small improvement we can reach a competitive ratio of 1.5787. Both these results beat the lower bound of 1.58333 by Ramanan et al. [95], which applied to all previously known algorithms, and reduce the gap to the general lower bound by more than 15% and more than 20%, respectively. The key to this improvement was to introduce new analytical tools to keep track of the existing items and their packing, so that we could improve the bounds given by the standard weighting function based analysis. Essentially, we could identify three different packing structures that are all advantageous to our algorithm in a different way, and we could combine all of these cases in one unified analysis by using a marking scheme and extending the linear program bounding the competitive ratio. Furthermore, we deviated from previous more restrictive approaches in some situations where we could anticipate that this deviation would be helpful.

We also studied lower bounds for general algorithms as well as for several special algorithm classes for two-dimensional online bin packing. This analysis shows that our ideas from the one-dimensional case do not suffice to improve upon previous, more restricted algorithms like Harmonic-type algorithms. There is, however, still some gap to the general lower bound, which means that new ideas are needed in order to obtain a better two-dimensional algorithm.

In the geometric knapsack problem, we reduce the running time of known PTAS results from $\Omega\left(n^{2^{2^{1/\varepsilon}}}\right)$ to $O_\varepsilon(1) \cdot n^{O(1)}$. Thus, we obtain an EPTAS which is the best result we can hope for[1], as an FPTAS cannot exist for this strongly NP-complete problem unless P = NP [87]. In essence, we use careful examination of the input items to make this improvement possible. The crucial observation was that for identifying some interesting items in the input, we do not need to consider all input items as candidates and guess the right ones directly, but we can reduce the running time significantly by chosing the candidates more cleverly. To this end, we *first* analyze how much profit different choices could give us and *then* select candidate values for the guessing using this information. That way, we obtained an EPTAS for two-dimensional geometric knapsack with square items. In addition, by making use of resource augmentation, we could give an *optimal* algorithm with EPTAS running time for two-dimensional geometric knapsack with rectangular items.

Although these problems obviously differ in their methods and techniques, the key to most of our results is to leverage knowledge about the input structure in

[1] up to improving the constant factors

order to improve previous results. While this is a straightforward idea, it can be very challenging to understand *how* to incorporate such structural knowledge into algorithms and their analysis. This thesis shows that this is indeed possible and facilitates significantly improved algorithms. Thus we hope that some of our results and techniques give inspiration for further advances in this field.

Appendices

A Parameters for a 1.583-Competitive Extreme Harmonic-Algorithm

We provide parameters of an Extreme Harmonic-algorithm with a competitive ratio of 1.583. These are provided to show that the Ramanan et al. [95] lower bound of 1.58333 can be beaten with much simpler input that also produces very few knapsack problems only (indeed 23 knapsack problems need to be considered). Thus, we hope that this enables a reader to check rather easily that we can indeed break this lower bound.

First, let us provide the input given to our parameter optimization program (i.e., the manual input without automatically generated types) in table A.1.

Parameter	Value
c	$\frac{1583}{1000}$
t_N	$\frac{1}{100}$
Γ	$\frac{2}{7}$ (starting from $\frac{1}{12}$)
$\mathcal{T}$	$\frac{1}{30}$

(a) Parameters

Item size	red_i
335/1000	0
334/1000	0
5/18	2/100
7/27	105/1000
1/4	106/1000
8/39	8/100
1/5	93/1000
3/17	3/100
1/6	8/100
3/20	0
29/200	0
1/7	135/1000
1/13	1/10
1/14	1/13

(b) Size lower bounds and initial values red_i

Table A.1: Parameters and item types.

In table A.2, we again list some excerpt of the complete list of types with their parameters $\text{red}_i, \text{redfit}_i, \text{needs}(i)$, and $\text{leaves}(i)$. For $i > 42$, we have $t_i = 1/(i-26)$.

Type i	t_i	red_i	redfit_i	needs(i)	leaves(i)
3	1/2	0	0	0	0
4	41783/100000	≈ 0.0158	1	23	0
5	41/100	≈ 0.0360	1	22	2
6	40/100	≈ 0.0562	1	21	3
7	39/100	≈ 0.0764	1	20	4

Type i	t_i	red_i	redfit_i	needs(i)	leaves(i)
8	38/100	≈ 0.0966	1	19	6
9	37/100	≈ 0.1168	1	18	7
10	36/100	≈ 0.1370	1	17	8
11	35/100	≈ 0.1572	1	16	9
12	34/100	≈ 0.1673	1	15	10
13	335/1000	≈ 0.1694	1	14	11
14	334/1000	≈ 0.1707	1	13	11
15	1/3	0	0	0	0
16	$29/90 \approx 0.3222$	0	0	0	0
17	$11/36 \approx 0.3056$	≈ 0.0200	1	10	0
18	$5/18 \approx 0.2778$	≈ 0.1050	1	8	1
19	$7/27 \approx 0.2593$	≈ 0.1060	1	7	5
20	1/4	≈ 0.0800	1	7	0
21	$8/39 \approx 0.2051$	≈ 0.0930	1	4	2
22	1/5	≈ 0.0300	1	3	0
23	$3/17 \approx 0.1765$	≈ 0.0800	1	2	0
24	$1/6 \approx 0.1667$	≈ 0.0333	1	1	0
25	$29/180 \approx 0.1611$	≈ 0.0833	2	11	0
26	$11/72 \approx 0.1528$	≈ 0.1000	2	10	0
27	15/100	≈ 0.1300	2	9	0
28	145/1000	≈ 0.1429	2	9	0
29	$1/7 \approx 0.1429$	≈ 0.1250	2	9	0
30	$1/8 = 0.125$	≈ 0.1111	2	7	0
31	$1/9 \approx 0.1111$	≈ 0.0333	2	5	0
32	$29/270 \approx 0.1074$	≈ 0.0833	3	11	0
33	$11/108 =\approx 0.1019$	≈ 0.1000	3	10	0
34	$1/10 = 0.1$	≈ 0.0909	3	9	0
35	$1/11 \approx 0.0909$	≈ 0.0833	3	8	0
36	$1/12 \approx 0.0833$	≈ 0.0333	3	7	0
37	$29/360 \approx 0.0806$	≈ 0.1231	3	7	0
38	$1/13 \approx 0.0769$	≈ 0.0566	3	6	0
39	$11/144 \approx 0.0764$	≈ 0.1179	3	6	0
40	$1/14 \approx 0.0714$	≈ 0.1133	4	9	0
41	$1/15 \approx 0.0667$	≈ 0.1094	4	8	0
42	$1/16 = 0.0625$	≈ 0.1059	4	7	0
		$\vdots$			
123	$1/97 \approx 0.0103$	≈ 0.0107	27	8	0
124	$1/98 \approx 0.0102$	≈ 0.0106	28	9	0
125	$1/99 \approx 0.0101$	≈ 0.0105	28	9	0

Table A.2: Parameters of a 1.583-competitive Extreme Harmonic-algorithm.

We give a list of all redspace-values that are at most 1/3 in table A.3. The redspace-values above 1/3 are equal to the t_i-values above 1/3.

Finally, there were only two different y_3^*-values used to establish the feasibility of the dual LPs: $y_3^* = 9/32$ for the cases $k = 2, 3, 4, 6, 7, 9, 10, 11$ and $y_3^* = 3/16$ in all other cases.

index i	redspace$_i$	index i	redspace$_i$	index i	redspace$_i$
0	0	4	11/50	8	7/25
1	1/6	5	2/9	9	3/10
2	3/17	6	6/25	10	8/25
3	1/5	7	13/50	11	33/100

Table A.3: redspace-values below 1/3 in the 1.583-competitive algorithm.

B Improved Parameters for a 1.5884-Competitive Super Harmonic-Algorithm

Finally, we want to give some improved parameters for the Super Harmonic framework in order to show that a competitive ratio of 1.5884 is possible for this framework. In particular, the parameters used here are much simpler than the ones used by Seiden (which were manually optimized up to a precision of 10^{-7}). The type thresholds are the same, we mainly optimized the red_i-values. Also note that the redfit_i-values for small types with $t_i > 1/18$ are computed differently than in Harmonic++: we have $\text{redfit}_i = \lfloor \frac{24/83}{t_i} \rfloor$ (unless $\text{red}_i = 0$, in which case of course $\text{redfit}_i = 0$).

Type i	t_i	red_i	redfit_i	needs(i)	leaves(i)
1	1	0	0	0	0
2	$341/512 \approx 0.6660$	0	0	0	13
3	$511/768 \approx 0.6654$	0	0	0	14
4	$85/128 \approx 0.6641$	0	0	0	15
5	$127/192 \approx 0.6615$	0	0	0	16
6	$21/32 \approx 0.6563$	0	0	0	17
7	$31/48 \approx 0.6458$	0	0	0	18
8	$5/8 = 0.625$	0	0	0	19
9	$7/12 \approx 0.5833$	0	0	0	20
10	1/2	0	0	0	0
11	$5/12 \approx 0.4167$	≈ 0.0900	1	20	4
12	$3/8 = 0.375$	≈ 0.1335	1	19	6
13	$17/48 \approx 0.3542$	≈ 0.1555	1	18	7
14	$11/32 \approx 0.3438$	≈ 0.1658	1	17	8
15	$65/192 \approx 0.3385$	≈ 0.1712	1	16	9
16	$43/128 \approx 0.3359$	≈ 0.1740	1	15	10
17	$257/768 \approx 0.3346$	≈ 0.1750	1	14	11
18	$171/512 \approx 0.3340$	≈ 0.1754	1	13	12
19	1/3	0	0	0	0
20	$13/48 \approx 0.2708$	≈ 0.0925	1	7	5
21	$1/4 = 0.25$	≈ 0.0736	1	6	0
22	$13/63 \approx 0.2063$	≈ 0.1000	1	6	5
23	$1/5 = 0.2$	≈ 0.0350	1	6	0
24	$15/88 \approx 0.1705$	≈ 0.0830	1	5	3
25	$1/6 \approx 0.1667$	≈ 0.0789	1	4	0
26	$12/83 \approx 0.1446$	≈ 0.1300	2	7	2
27	$1/7 \approx 0.1429$	≈ 0.0145	2	7	0
28	$11/83 \approx 0.1325$	≈ 0.0710	2	7	1
29	$1/8 = 0.125$	≈ 0.0596	2	6	0
30	$1/9 \approx 0.1111$	≈ 0.0500	2	6	0

Type i	t_i	red_i	redfit_i	needs(i)	leaves(i)
31	$1/10 = 0.1$	≈ 0.0450	2	6	0
32	$1/11 \approx 0.0909$	≈ 0.0320	3	7	0
33	$1/12 \approx 0.0833$	≈ 0.0220	3	6	0
34	$1/13 \approx 0.0769$	≈ 0.0355	3	6	0
35	$1/14 \approx 0.0714$	≈ 0.0085	4	7	0
36	$1/15 \approx 0.0667$	≈ 0.0100	4	7	0
37	$1/16 = 0.0625$	≈ 0.0100	4	6	0
38	$1/17 \approx 0.0588$	≈ 0.0100	4	6	0
39	$1/18 \approx 0.0556$	0	0	0	0
		$\vdots$			
68	$1/47 \approx 0.0213$	0	0	0	0
69	$1/48 \approx 0.0208$	0	0	0	0

Table B.1: Parameters used for our improvement of Harmonic++.

C Parameters for a Competitive Ratio of 1.5787

In this chapter, we want to give parameters for a Extreme Harmonic algorithm that uses red sand as described in section 4.6 and that achieves a competitive ratio of 1.5787. We start by giving the redspace-values; these are mostly defined by the types (see table C.3).

Table C.1: redspace-values for the 1.5787-competitive algorithm.

k	redspace_k
1	$1 - 2 \cdot t_{22} = 89/500$
2…22	$1 - 2 \cdot t_{22+k} = (89+k)/500$
23	2/9
24…236	$1 - 2 \cdot t_{21+k}$
237…491	t_{495-k}

We list the y_1^*, y_2^*, and y_3^* values in table C.2.

For the types, we only list type thresholds below 1/2 in table C.3, as types above this are determined depending on the case as described in section 4.2.7. For type $N = 753$, we have $\text{red}_N = 3/20$ and $\text{redspace}_{\text{needs}(N)} = 17/60$. For many of the medium types, the parameter red_i is determined by the following linear function $f(i) = \frac{18977}{25800} - \frac{368}{215} \cdot t_{i+1}$.

Type i	t_i	red_i	redfit_i	needs(i)	leaves(i)
3	1/2	0	0	0	0
4…21	$\frac{429}{1000} - \frac{i-4}{1000}$	1183/10000	1	$495 - i$	0
22	411/1000	1183/10000	1	473	1
23…44	$\frac{41}{100} - \frac{i-23}{1000}$	1183/10000	1	$495 - i$	$i - 22$
45…74	$\frac{41}{100} - \frac{i-23}{1000}$	1183/10000	1	$495 - i$	$i - 21$
75	43/120	311/2500	1	420	54
76	179/500	311/2500	1	419	55
77	357/1000	311/2500	1	418	56
78	89/250	311/2500	1	417	57
79	59/166	1339/10000	1	416	58
80…84	$\frac{71}{200} - \frac{i-80}{1000}$	1339/10000	1	$495 - i$	$i - 21$
85…94	$\frac{7}{20} - \frac{i-85}{9600}$	71/500	1	$495 - i$	$i - 21$
95	349/1000	71/500	1	400	74
96…105	$\frac{67}{192} - \frac{i-96}{9600}$	71/500	1	$495 - i$	$i - 21$
106	87/250	71/500	1	389	85
107…115	$\frac{167}{480} - \frac{i-107}{9600}$	71/500	1	$495 - i$	$i - 21$

Type i	t_i	red_i	redfit_i	needs(i)	leaves(i)
116	347/1000	71/500	1	379	95
117	3331/9600	71/500	1	378	96
118	111/320	71/500	1	377	97
119…126	$\frac{3329}{9600} - \frac{i-119}{9600}$	33/200	1	$495 - i$	$i - 21$
127	173/500	33/200	1	368	106
128…146	$\frac{1107}{3200} - \frac{i-128}{9600}$	33/200	1	$495 - i$	$i - 21$
147	43/125	33/200	1	348	126
148…157	$\frac{1651}{4800} - \frac{i-148}{9600}$	33/200	1	$495 - i$	$i - 21$
158	343/1000	33/200	1	337	137
159…163	$\frac{823}{2400} - \frac{i-159}{9600}$	33/200	1	$495 - i$	$i - 21$
164…167	$\frac{823}{2400} - \frac{i-159}{9600}$	$f(i)$	1	$495 - i$	$i - 21$
168	171/500	$f(i)$	1	327	147
169…178	$\frac{3283}{9600} - \frac{i-169}{9600}$	$f(i)$	1	$495 - i$	$i - 21$
179	341/1000	$f(i)$	1	316	158
180…198	$\frac{1091}{3200} - \frac{i-180}{9600}$	$f(i)$	1	$495 - i$	$i - 21$
199	339/1000	$f(i)$	1	296	178
200…209	$\frac{1627}{4800} - \frac{i-200}{9600}$	$f(i)$	1	$495 - i$	$i - 21$
210	169/500	$f(i)$	1	285	189
211…219	$\frac{811}{2400} - \frac{i-211}{9600}$	$f(i)$	1	$495 - i$	$i - 21$
220	337/1000	$f(i)$	1	275	199
221…230	$\frac{647}{1920} - \frac{i-221}{9600}$	$f(i)$	1	$495 - i$	$i - 21$
231	42/125	$f(i)$	1	264	210
232…250	$\frac{43}{128} - \frac{i-232}{9600}$	$f(i)$	1	$495 - i$	$i - 21$
251	167/500	$f(i)$	1	244	230
252	1603/4800	$f(i)$	1	243	231
253	641/1920	$f(i)$	1	242	232
254	267/800	$f(i)$	1	241	233
255	3203/9600	$f(i)$	1	240	234
256	1601/4800	$f(i)$	1	239	235
257	1067/3200	$f(i)$	1	238	236
258	1/3	0	0	0	0
259	49/150	0	0	0	0
260	19/60	1/50	1	151	0
261	5/18	21/200	1	51	0
262	7/27	53/500	1	42	23
263	23/90	93/1000	1	40	28
264	63/250	97/1000	1	38	34
265	1/4	7/100	1	37	0
266	2/9	2/25	1	23	0
267	8/39	211/2500	1	14	1
268	1/5	3/100	1	11	0
269	3/17	143/10000	1	1	0
270…275	$1/2 - t_{528-i}$	29/250	2	$507 - i$	0
276…279	$1/2 - t_{528-i}$	237/2000	2	$507 - i$	0
280…356	$1/2 - t_{528-i}$	11949/100000	2	$507 - i$	0
357	633/4000	11949/100000	2	151	0
358…443	$1/2 - t_{529-i}$	11949/100000	2	$508 - i$	0

Type i	t_i	red_i	redfit_i	needs(i)	leaves(i)
444	3/20	4/25	2	64	0
445	599/4000	4/25	2	64	0
446…450	$1/2 - t_{530-i}$	4/25	2	$509 - i$	0
451	579/4000	4/25	2	59	0
452…454	$1/2 - t_{531-i}$	167/1000	2	$510 - i$	0
455	1/7	971/10000	2	56	0
456…465	$1/2 - t_{532-i}$	971/1000	2	$511 - i$	0
466…473	$1/2 - t_{532-i}$	18/125	2	$511 - i$	0
474…481	$1/2 - t_{532-i}$	33/250	2	$511 - i$	0
482	471/4000	33/250	2	30	0
483…488	$1/2 - t_{533-i}$	33/250	2	$512 - i$	0
489	1/9	3/25	2	23	0
490…492	$1/2 - t_{534-i}$	3/25	2	$512 - i$	0
493	49/450	3/25	2	20	0
494…496	$1/2 - t_{535-i}$	3/25	2	$513 - i$	0
497	19/180	3/25	2	17	0
498…502	$1/2 - t_{536-i}$	3/25	2	$514 - i$	0
503…512	$1/2 - t_{536-i}$	109/1000	2	$514 - i$	0
513	1/11	491/5000	3	49	0
514…517	$1/2 - t_{537-i}$	491/5000	3	$561 - i$	0
518…519	$1/2 - t_{537-i}$	491/5000	3	$559 - i$	0
520	$1/2 - t_{17}$	491/5000	3	38	0
521	1/12	31/250	3	37	0
522	$1/2 - t_{16}$	31/250	3	37	0
523	$1/2 - t_{15}$	31/250	3	35	0
524	49/600	31/250	3	35	0
525	$1/2 - t_{14}$	31/250	3	34	0
526	$1/2 - t_{13}$	31/250	3	32	0
527	19/240	31/250	3	31	0
528	$1/2 - t_{12}$	31/250	3	31	0
529	$1/2 - t_{11}$	31/250	3	29	0
530	$1/2 - t_{10}$	31/250	3	28	0
531	1/13	127/1000	3	28	0
532	19/250	127/1000	3	26	0
533	3/40	127/1000	3	25	0
534	37/500	127/1000	3	22	0
535	73/1000	127/1000	3	21	0
536	9/125	127/1000	3	19	0
537	1/14	1/10	3	19	0
538	71/1000	1/10	3	18	0
539	1/15	1089/10000	4	46	0
540	1/16	1147/10000	4	37	0
541	1/17	1071/10000	4	30	0
542	1/18	1019/10000	5	51	0
543	1/19	269/2500	5	44	0
544	1/20	1/10	5	37	0
545	1/21	1/10	5	32	0
546	1/22	123/1250	6	49	0

Type i	t_i	red_i	redfit_i	needs(i)	leaves(i)
547	1/23	233/2500	6	43	0
548	1/24	1023/10000	6	37	0
549	1/25	987/10000	7	52	0
550	1/26	59/625	7	47	0
551	1/27	91/1000	7	42	0
552	1/28	871/10000	7	37	0
553	1/29	1001/10000	8	50	0
554	1/30	49/400	8	46	0
555	1/31	263/2000	8	42	0
556	1/32	903/10000	9	53	0
557	1/33	871/10000	9	49	0
558	1/34	21/250	9	45	0
559	1/35	83/1000	9	41	0
560	1/36	41/500	10	51	0
561	1/37	41/500	10	48	0
562	1/38	173/2000	10	44	0
563	1/39	421/5000	11	54	0
564	1/40	893/10000	11	50	0
565	1/41	87/1000	11	47	0
566	1/42	849/10000	11	43	0
567	1/43	849/10000	12	52	0
568	1/44	849/10000	12	49	0
569	1/45	849/10000	12	46	0
570	1/46	849/10000	12	43	0
571	1/47	849/10000	13	51	0
572	1/48	849/10000	13	48	0
573	1/49	849/10000	13	45	0
574	1/50	3479/19980	14	52	0
575	1/51	10031/57720	14	50	0
576	1/52	61229/352980	14	47	0
577	1/53	7784/44955	14	45	0
578	1/54	1407/8140	15	51	0
579	1/55	4597/26640	15	49	0
580	1/56	65401/379620	15	46	0
581	1/57	5537/32190	15	44	0
582	1/58	67487/392940	16	50	0
583	1/59	6853/39960	16	48	0
584	1/60	23191/135420	16	46	0
585	1/61	8827/51615	17	52	0
586	1/62	10237/59940	17	50	0
587	1/63	4039/23680	17	47	0
588	1/64	14749/86580	17	45	0
589	1/65	18697/109890	18	51	0
590	1/66	25277/148740	18	49	0
591	1/67	2261/13320	18	47	0
592	1/68	77917/459540	19	52	0
593	1/69	94/555	19	50	0
594	1/70	80003/472860	19	48	0

Type i	t_i	red_i	redfit_i	needs(i)	leaves(i)
595	1/71	40523/239760	19	46	0
596	1/72	9121/54020	20	51	0
597	1/73	20783/123210	20	49	0
598	1/74	91/540	20	48	0
599	1/75	14203/84360	21	52	0
600	1/76	12323/73260	21	51	0
601	1/77	10913/64935	21	49	0
602	1/78	29449/175380	21	47	0
603	1/79	8939/53280	22	52	0
604	1/80	90433/539460	22	50	0
605	1/81	2541/15170	22	48	0
606	1/82	92519/552780	22	47	0
607	1/83	6683/39960	23	51	0
608	1/84	371/2220	23	49	0
609	1/85	11956/71595	23	48	0
610	1/86	96691/579420	24	52	0
611	1/87	16289/97680	24	50	0
612	1/88	98777/592740	24	49	0
613	1/89	4991/29970	24	47	0
614	1/90	1601/9620	25	51	0
615	1/91	50953/306360	25	50	0
616	1/92	102949/619380	25	48	0
617	1/93	4333/26085	26	52	0
618	1/94	21007/126540	26	51	0
619	1/95	53039/319680	26	49	0
620	1/96	35707/215340	26	48	0
621	1/97	3863/23310	27	52	0
622	1/98	109207/659340	27	50	0
623	1/99	49/296	27	49	0
624	1/100	3479/19980	28	52	0
625	1/102	10031/57720	28	50	0
626	1/104	61229/352980	29	52	0
627	1/106	7784/44955	29	49	0
628	1/108	1407/8140	30	51	0
629	1/110	4597/26640	30	49	0
630	1/112	65401/379620	31	51	0
631	1/114	5537/32190	31	48	0
632	1/116	67487/392940	32	50	0
633	1/118	6853/39960	33	52	0
634	1/120	23191/135420	33	50	0
635	1/122	8827/51615	34	52	0
636	1/124	10237/59940	34	50	0
637	1/126	4039/23680	35	51	0
638	1/128	14749/86580	35	49	0
639	1/130	18697/109890	36	51	0
640	1/132	25277/148740	36	49	0
641	1/134	2261/13320	37	51	0
642	1/136	77917/459540	38	52	0

Type i	t_i	red_i	redfit_i	needs(i)	leaves(i)
643	1/138	94/555	38	50	0
644	1/140	80003/472860	39	52	0
645	1/142	40523/239760	39	50	0
646	1/144	9121/54020	40	51	0
647	1/146	20783/123210	40	49	0
648	1/148	91/540	41	51	0
649	1/150	3479/19980	42	52	0
650	1/153	10031/57720	42	50	0
651	1/156	61229/352980	43	50	0
652	1/159	7784/44955	44	51	0
653	1/162	1407/8140	45	51	0
654	1/165	4597/26640	46	52	0
655	1/168	65401/379620	47	52	0
656	1/171	5537/32190	47	50	0
657	1/174	67487/392940	48	50	0
658	1/177	6853/39960	49	51	0
659	1/180	23191/135420	50	51	0
660	1/183	8827/51615	51	52	0
661	1/186	10237/59940	52	52	0
662	1/189	4039/23680	52	50	0
663	1/192	14749/86580	53	51	0
664	1/195	18697/109890	54	51	0
665	1/198	25277/148740	55	51	0
666	1/201	15841/91020	56	52	0
667	1/205	241787/1391940	57	52	0
668	1/209	245959/1418580	58	51	0
669	1/213	11911/68820	59	51	0
670	1/217	14959/86580	60	51	0
671	1/221	10339/59940	61	51	0
672	1/225	29183/169460	63	52	0
673	1/229	266819/1551780	64	52	0
674	1/233	270991/1578420	65	52	0
675	1/237	91721/535020	66	52	0
676	1/241	7981/46620	67	52	0
677	1/245	283507/1658340	68	51	0
678	1/249	95893/561660	69	51	0
679	1/253	74711/429570	70	51	0
680	1/258	101353/583860	72	52	0
681	1/263	154637/892440	73	51	0
682	1/268	44927/259740	75	52	0
683	1/273	13321/77145	76	52	0
684	1/278	324919/1884780	77	51	0
685	1/283	165067/959040	79	52	0
686	1/288	37261/216820	80	51	0
687	1/293	85141/496170	82	52	0
688	1/298	345779/2017980	83	52	0
689	1/303	119329/685980	84	51	0
690	1/309	3469/19980	86	52	0

Type i	t_i	red_i	redfit_i	needs(i)	leaves(i)
691	1/315	41167/237540	88	52	0
692	1/321	125587/725940	89	51	0
693	1/327	127673/739260	91	52	0
694	1/333	1169/6780	93	52	0
695	1/339	26369/153180	94	51	0
696	1/345	133931/779220	96	52	0
697	1/351	11529/66230	98	52	0
698	1/358	84469/486180	100	52	0
699	1/365	214823/1238760	102	52	0
700	1/372	145649/841380	104	52	0
701	1/379	55531/321345	106	52	0
702	1/386	451549/2617380	108	52	0
703	1/393	3059/17760	110	52	0
704	1/400	3479/19980	112	52	0
705	1/408	10031/57720	114	52	0
706	1/416	61229/352980	116	52	0
707	1/424	7784/44955	118	52	0
708	1/432	1407/8140	120	51	0
709	1/440	4597/26640	123	52	0
710	1/448	65401/379620	125	52	0
711	1/456	35903/206460	127	52	0
712	1/465	45661/263070	130	52	0
713	1/474	26539/153180	132	52	0
714	1/483	94451/546120	135	52	0
715	1/492	192031/1112220	137	52	0
716	1/501	28213/162060	140	52	0
717	1/511	602903/3469860	143	52	0
718	1/521	613333/3536460	145	52	0
719	1/531	69307/400340	148	52	0
720	1/541	634193/3669660	151	52	0
721	1/551	81452/467865	154	52	0
722	1/562	663089/3816180	157	52	0
723	1/573	112427/648240	160	52	0
724	1/584	1153/6660	163	52	0
725	1/595	174377/1008990	166	52	0
726	1/606	119329/685980	169	52	0
727	1/618	3469/19980	173	52	0
728	1/630	41167/237540	176	52	0
729	1/642	125587/725940	179	52	0
730	1/654	257677/1480740	183	52	0
731	1/667	4627/26640	186	52	0
732	1/680	114307/659340	190	52	0
733	1/693	22603/130610	194	52	0
734	1/706	41713/239760	197	52	0
735	1/720	47159/271580	201	52	0
736	1/734	6349/36630	205	52	0
737	1/748	439033/2537460	209	52	0
738	1/762	42841/246420	213	52	0

Type i	t_i	red_i	redfit_i	needs(i)	leaves(i)
739	1/777	4123/23760	217	52	0
740	1/792	103439/597180	221	52	0
741	1/807	317863/1827060	225	52	0
742	1/823	970277/5587740	230	52	0
743	1/839	197393/1138860	234	52	0
744	1/855	56147/322640	239	52	0
745	1/872	146911/845820	244	52	0
746	1/889	261527/1508490	248	52	0
747	1/906	3187/18315	253	52	0
748	1/924	181601/1045620	258	52	0
749	1/942	18473/106560	263	52	0
750	1/960	378049/2173380	268	52	0
751	1/979	288491/1661670	274	52	0
752	1/998	10549/66600	279	52	0

Table C.3: Parameters for the 1.5787 algorithm.

Table C.2: y_1^*-, y_2^*- and y_3^*-values for the 1.5787-competitive algorithm. In cases where no y_1^*-value is given, $D_w^{k,\text{sml}}$ was used. Note that only three different values for y_3^* were used.

k	$y_1^* = \frac{y_2^*}{2}$	y_3^*
≤ 229	–	$\frac{1}{4}$
$230 \dots 237$	–	$\frac{7}{32}$
$238 \dots 243$	$\frac{37000301}{10153836000}$	$\frac{3}{16}$
244	$\frac{4515397}{1269229500}$	$\frac{3}{16}$
245	$\frac{12138517}{3384612000}$	$\frac{3}{16}$
$246 \dots 263$	$\frac{37000301}{10153836000}$	$\frac{3}{16}$
264	$\frac{12138517}{3384612000}$	$\frac{3}{16}$
265	$\frac{4515397}{1269229500}$	$\frac{3}{16}$
$266 \dots 274$	$\frac{37000301}{10153836000}$	$\frac{3}{16}$
275	$\frac{35830801}{10153836000}$	$\frac{1}{4}$
276	$\frac{18353963}{5076918000}$	$\frac{3}{16}$
$277 \dots 284$	$\frac{37000301}{10153836000}$	$\frac{1}{4}$
285	$\frac{18353963}{5076918000}$	$\frac{1}{4}$
286	$\frac{35830801}{10153836000}$	$\frac{1}{4}$
$287 \dots 295$	$\frac{37000301}{10153836000}$	$\frac{1}{4}$
296	$\frac{4515397}{1269229500}$	$\frac{1}{4}$
297	$\frac{12138517}{3384612000}$	$\frac{1}{4}$
$298 \dots 315$	$\frac{37000301}{10153836000}$	$\frac{1}{4}$
316	$\frac{12138517}{3384612000}$	$\frac{1}{4}$
317	$\frac{4515397}{1269229500}$	$\frac{1}{4}$
$318 \dots 326$	$\frac{37000301}{10153836000}$	$\frac{1}{4}$
327	$\frac{35830801}{10153836000}$	$\frac{1}{4}$
328	$\frac{18353963}{5076918000}$	$\frac{1}{4}$
$329 \dots 336$	$\frac{37000301}{10153836000}$	$\frac{1}{4}$
337	$\frac{18353963}{5076918000}$	$\frac{1}{4}$
338	$\frac{35830801}{10153836000}$	$\frac{1}{4}$
$339 \dots 347$	$\frac{37000301}{10153836000}$	$\frac{1}{4}$
348	$\frac{4515397}{1269229500}$	$\frac{1}{4}$
349	$\frac{12138517}{3384612000}$	$\frac{1}{4}$
$350 \dots 367$	$\frac{37000301}{10153836000}$	$\frac{1}{4}$
368	$\frac{12138517}{3384612000}$	$\frac{1}{4}$
369	$\frac{4515397}{1269229500}$	$\frac{1}{4}$
$370 \dots 378$	$\frac{37000301}{10153836000}$	$\frac{1}{4}$
379	$\frac{35830801}{10153836000}$	$\frac{1}{4}$
380	$\frac{18353963}{5076918000}$	$\frac{1}{4}$
$381 \dots 388$	$\frac{37000301}{10153836000}$	$\frac{1}{4}$
389	$\frac{18353963}{5076918000}$	$\frac{1}{4}$
390	$\frac{35830801}{10153836000}$	$\frac{1}{4}$
$391 \dots 399$	$\frac{37000301}{10153836000}$	$\frac{1}{4}$
400	$\frac{4515397}{1269229500}$	$\frac{1}{4}$
401	$\frac{12138517}{3384612000}$	$\frac{1}{4}$
$402 \dots 410$	$\frac{37000301}{10153836000}$	$\frac{1}{4}$
$411 \dots 491$	–	$\frac{1}{4}$

Bibliography

[1] Anna Adamaszek and Andreas Wiese. "A quasi-PTAS for the Two-Dimensional Geometric Knapsack Problem." In: *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2015)*. SIAM, 2015, pp. 1491–1505. DOI: 10.1137/1.9781611973730.98.

[2] Avenir Software Inc. *LengthWise 2014*. http://www.avenir-online.com/AvenirWeb/Lengthwise/LengthWiseHome.aspx.

[3] Luitpold Babel, Bo Chen, Hans Kellerer, and Vladimir Kotov. "On-Line Algorithms for Cardinality Constrained Bin Packing Problems." In: *Algorithms and Computation: 12th International Symposium, ISAAC 2001 Christchurch, New Zealand, December 19–21, 2001 Proceedings*. Ed. by Peter Eades and Tadao Takaoka. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 695–706. ISBN: 978-3-540-45678-0. DOI: 10.1007/3-540-45678-3_59.

[4] Luitpold Babel, Bo Chen, Hans Kellerer, and Vladimir Kotov. "Algorithms for on-line bin-packing problems with cardinality constraints." In: *Discrete Applied Mathematics* 143.1-3 (2004), pp. 238–251. DOI: 10.1016/j.dam.2003.05.006.

[5] Brenda S. Baker and Edward G. Coffman Jr. "A tight asymptotic bound for next-fit-decreasing bin-packing." In: *SIAM Journal on Algebraic Discrete Methods* 2.2 (1981), pp. 147–152.

[6] Brenda S. Baker, A. Robert Calderbank, Edward G. Coffman Jr., and Jeffrey C. Lagarias. "Approximation Algorithms for Maximizing the Number of Squares Packed into a Rectangle." In: *SIAM Journal on Algebraic Discrete Methods* 4.3 (1983), pp. 383–397. DOI: 10.1137/0604039.

[7] János Balogh, József Békési, György Dósa, Leah Epstein, and Asaf Levin. "Lower bounds for several online variants of bin packing." In: *Approximation and Online Algorithms - 15th International Workshop, WAOA 2017, Proceedings*, To appear. arXiv: 1708.03228v1 [cs.DS].

[8] János Balogh, József Békési, and Gábor Galambos. "New lower bounds for certain classes of bin packing algorithms." In: *Theoretical Computer Science* 440-441 (2012), pp. 1–13. DOI: 10.1016/j.tcs.2012.04.017.

[9] János Balogh, József Békési, György Dósa, Jirí Sgall, and Rob van Stee. "The optimal absolute ratio for online bin packing." In: *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*. Ed. by Piotr Indyk. SIAM, 2015, pp. 1425–1438. ISBN: 978-1-61197-374-7. DOI: 10.1137/1.9781611973730.94.

[10] János Balogh, József Békési, György Dósa, Leah Epstein, and Asaf Levin. "A new and improved algorithm for online bin packing." In: *CoRR* abs/1707.01728 (2017). arXiv: 1707.01728.

[11] János Balogh, József Békési, György Dósa, Leah Epstein, and Asaf Levin. "Online Bin Packing with Cardinality Constraints Resolved." In: *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*. Ed. by Kirk Pruhs and Christian Sohler. Vol. 87. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 10:1–10:14. ISBN: 978-3-95977-049-1. DOI: 10.4230/LIPIcs.ESA.2017.10.

[12] Nikhil Bansal and Arindam Khan. "Improved Approximation Algorithm for Two-Dimensional Bin Packing." In: *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*. Ed. by Chandra Chekuri. SIAM, 2014, pp. 13–25. ISBN: 978-1-61197-338-9. DOI: 10.1137/1.9781611973402.2.

[13] Nikhil Bansal, Zhen Liu, and Arvind Sankar. "Bin-Packing with Fragile Objects." In: *Foundations of Information Technology in the Era of Networking and Mobile Computing, IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002), August 25-30, 2002, Montréal, Québec, Canada*. Ed. by Ricardo A. Baeza-Yates, Ugo Montanari, and Nicola Santoro. Vol. 223. IFIP Conference Proceedings. Kluwer, 2002, pp. 38–46. ISBN: 1-4020-7181-7.

[14] Nikhil Bansal, José R. Correa, Claire Kenyon, and Maxim Sviridenko. "Bin Packing in Multiple Dimensions: Inapproximability Results and Approximation Schemes." In: *Mathematics of Operations Research* 31.1 (2006), pp. 31–49. DOI: 10.1287/moor.1050.0168.

[15] Nikhil Bansal, Alberto Caprara, and Maxim Sviridenko. "A New Approximation Method for Set Covering Problems, with Applications to Multidimensional Bin Packing." In: *SIAM Journal on Computing* 39.4 (2009), pp. 1256–1278. DOI: 10.1137/080736831.

[16] Nikhil Bansal, Alberto Caprara, Klaus Jansen, Lars Prädel, and Maxim Sviridenko. "A Structural Lemma in 2-Dimensional Packing, and Its Implications on Approximability." In: *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*. Ed. by Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra. Vol. 5878. Lecture Notes in Computer Science. Springer, 2009, pp. 77–86. ISBN: 978-3-642-10630-9. DOI: 10.1007/978-3-642-10631-6_10.

[17] David Blitz. "Lower Bounds on the Asymptotic Worst-Case Ratios of On-line Bin Packing Algorithms." MA thesis. Erasmus Universiteit Rotterdam, 1995.

[18] David Blitz, Sandy Heydrich, Rob van Stee, André van Vliet, and Gerhard J. Woeginger. "Improved Lower Bounds for Online Hypercube and Rectangle Packing." In: *CoRR* abs/1607.01229 (2016). URL: http://arxiv.org/abs/1607.01229.

[19] Joan Boyar, Shahin Kamali, Kim S. Larsen, and Alejandro López-Ortiz. "Online Bin Packing with Advice." In: *Algorithmica* 74.1 (2016), pp. 507–527. DOI: 10.1007/s00453-014-9955-8.

[20] Joan Boyar, Lene M. Favrholdt, Christian Kudahl, Kim S. Larsen, and Jesper W. Mikkelsen. "Online Algorithms with Advice: A Survey." In: *ACM Computing Surveys* 50.2 (2017), 19:1–19:34. DOI: 10.1145/3056461.

[21] Donna J. Brown. *A lower bound for on-line one-dimensional bin packing algorithms*. Tech. rep. R-864. Coodinated Sci. Lab., University of Illinois at Urbana-Champaign, Urbana, 197.

[22] Alberto Caprara. "Packing 2-Dimensional Bins in Harmony." In: *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings*. IEEE Computer Society, 2002, pp. 490–499. ISBN: 0-7695-1822-2. DOI: 10.1109/SFCS.2002.1181973.

[23] Alberto Caprara and Michele Monaci. "On the two-dimensional Knapsack Problem." In: *Operations Research Letters* 32.1 (2004), pp. 5–14. DOI: 10.1016/S0167-6377(03)00057-9.

[24] Alberto Caprara, Hans Kellerer, Ulrich Pferschy, and David Pisinger. "Approximation algorithms for knapsack problems with cardinality constraints." In: *European Journal of Operational Research* 123.2 (2000), pp. 333–345. DOI: 10.1016/S0377-2217(99)00261-1.

[25] Miroslav Chlebík and Janka Chlebíková. "Hardness of approximation for orthogonal rectangle packing and covering problems." In: *Journal of Discrete Algorithms* 7.3 (2009), pp. 291–305. DOI: 10.1016/j.jda.2009.02.002.

[26] Marek Chrobak, Jiří Sgall, and Gerhard J. Woeginger. "Two-Bounded-Space Bin Packing Revisited." In: *Algorithms – ESA 2011: 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*. Ed. by Camil Demetrescu and Magnús M. Halldórsson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 263–274. ISBN: 978-3-642-23719-5. DOI: 10.1007/978-3-642-23719-5_23.

[27] Edward G. Coffman Jr., Michael R. Garey, and David S. Johnson. "An Application of Bin-Packing to Multiprocessor Scheduling." In: *SIAM Journal on Computing* 7.1 (1978), pp. 1–17. DOI: 10.1137/0207001.

[28] Edward G. Coffman Jr., Michael R. Garey, David S. Johnson, and Robert E. Tarjan. "Performance bounds for level-oriented two-dimensional packing algorithms." In: *SIAM Journal on Computing* 9 (1980), pp. 808–826. DOI: 10.1137/0209062.

[29] Edward G. Coffman Jr., Michael R. Garey, and David S. Johnson. "Approximation Algorithms for Bin-Packing — An Updated Survey." In: *Algorithm Design for Computer System Design*. Ed. by G. Ausiello, M. Lucertini, and P. Serafini. Vienna: Springer Vienna, 1984, pp. 49–106. ISBN: 978-3-7091-4338-4. DOI: 10.1007/978-3-7091-4338-4_3.

[30] Don Coppersmith and Prabhakar Raghavan. "Multidimensional on-line bin packing: Algorithms and worst-case analysis." In: *Operations Research Letters* 8.1 (1989), pp. 17 –20. ISSN: 0167-6377. DOI: 10.1016/0167-6377(89)90027-8.

[31] José R Correa and Claire Kenyon. "Approximation schemes for multidimensional packing." In: *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2004, pp. 186–195. URL: http://dl.acm.org/citation.cfm?id=982792.982820.

[32] János Csirik and David S. Johnson. "Bounded Space On-Line Bin Packing: Best Is Better than First." In: *Algorithmica* 31.2 (2001), pp. 115–138. DOI: 10.1007/s00453-001-0041-7.

[33] János Csirik and André van Vliet. "An on-line algorithm for multidimensional bin packing." In: *Operations Research Letters* 13.3 (1993), pp. 149 –158. ISSN: 0167-6377. DOI: 10.1016/0167-6377(93)90004-Z.

[34] George B. Dantzig. "Discrete-Variable Extremum Problems." In: *Operations Research* 5.2 (1957), pp. 266–288. DOI: 10.1287/opre.5.2.266.

[35] Kurt Eisemann. "The Trim Problem." In: *Management Science* 3.3 (1957), pp. 279–284. DOI: 10.1287/mnsc.3.3.279.

[36] Leah Epstein. "Two-dimensional online bin packing with rotation." In: *Theoretical Computer Science* 411.31-33 (2010), pp. 2899–2911. DOI: 10.1016/j.tcs.2010.04.021.

[37] Leah Epstein and Asaf Levin. "On Bin Packing with Conflicts." In: *Approximation and Online Algorithms: 4th International Workshop, WAOA 2006, Zurich, Switzerland, September 14-15, 2006. Revised Papers*. Ed. by Thomas Erlebach and Christos Kaklamanis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 160–173. ISBN: 978-3-540-69514-1. DOI: 10.1007/11970125_13.

[38] Leah Epstein and Rob van Stee. "Online square and cube packing." In: *Acta Informatica* 41.9 (2005), pp. 595–606. DOI: 10.1007/s00236-005-0169-z.

[39] Leah Epstein and Rob van Stee. "On Variable-Sized Multidimensional Packing." In: *Algorithms – ESA 2004: 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004. Proceedings*. Ed. by Susanne Albers and Tomasz Radzik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 287–298. ISBN: 978-3-540-30140-0. DOI: 10.1007/978-3-540-30140-0_27.

[40] Leah Epstein and Rob van Stee. "Optimal Online Algorithms for Multidimensional Packing Problems." In: *SIAM Journal on Computing* 35.2 (2005), pp. 431–448. DOI: 10.1137/S0097539705446895.

[41] Leah Epstein and Rob van Stee. "This side up!" In: *ACM Transactions on Algorithms* 2.2 (2006), pp. 228–243. DOI: 10.1145/1150334.1150339.

[42] Leah Epstein and Rob van Stee. "Bounds for online bounded space hypercube packing." In: *Discrete Optimization* 4.2 (June 2007), pp. 185–197. DOI: 10.1016/j.disopt.2006.11.005.

[43] Wenceslas Fernandez de la Vega and George S. Lueker. "Bin packing can be solved within $1 + \epsilon$ in linear time." In: *Combinatorica* 1.4 (1981), pp. 349–355. DOI: 10.1007/BF02579456.

[44] Aleksei Fishkin, Olga Gerber, Klaus Jansen, and Roberto Solis-Oba. "On Packing Rectangles with Resource Augmentation: Maximizing the Profit." In: *Algorithmic Operations Research* 3 (2008). URL: http://journals.hil.unb.ca/index.php/AOR/article/view/5664.

[45] Aleksei V. Fishkin, Olga Gerber, and Klaus Jansen. "On Efficient Weighted Rectangle Packing with Large Resources." In: *Algorithms and Computation, 16th International Symposium, ISAAC 2005, Sanya, Hainan, China, December 19-21, 2005, Proceedings*. Ed. by Xiaotie Deng and Ding-Zhu Du. Vol. 3827. Lecture Notes in Computer Science. Springer, 2005, pp. 1039–1050. ISBN: 3-540-30935-7. DOI: 10.1007/11602613_103.

[46] Aleksei V. Fishkin, Olga Gerber, Klaus Jansen, and Roberto Solis-Oba. "Packing Weighted Rectangles into a Square." In: *Mathematical Foundations of Computer Science 2005, 30th International Symposium, MFCS 2005, Gdansk, Poland, August 29 - September 2, 2005, Proceedings*. Ed. by Joanna Jedrzejowicz and Andrzej Szepietowski. Vol. 3618. Lecture Notes in Computer Science. Springer, 2005, pp. 352–363. ISBN: 3-540-28702-7. DOI: 10.1007/11549345_31.

[47] Ari Freund and Joseph Naor. "Approximating the Advertisement Placement Problem." In: *Journal of Scheduling* 7.5 (2004), pp. 365–374. DOI: 10.1023/B:JOSH.0000036860.90818.5f.

[48] Alan M. Frieze and Michael R. B. Clarke. "Approximation algorithms for the m-dimensional 0-1 knapsack problem: Worst-case and probabilistic analyses." In: *European Journal of Operational Research* 15.1 (1984), pp. 100–109. URL: https://ideas.repec.org/a/eee/ejores/v15y1984i1p100-109.html.

[49] Gábor Galambos. *A new heuristic for the classical bin packing problem*. Tech. rep. 82. Institut für Mathematik, Augsburg, 1985.

[50] Gábor Galambos and Gerhard J. Woeginger. "Repacking helps in bounded space on-line bind-packing." In: *Computing* 49.4 (1993), pp. 329–338. DOI: 10.1007/BF02248693.

[51] Waldo Gálvez, Fabrizio Grandoni, Sandy Heydrich, Salvatore Ingala, Arindam Khan, and Andreas Wiese. "Approximating Geometric Knapsack via L-Packings." In: *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*. Ed. by Chris Umans. IEEE Computer Society, 2017, pp. 260–271. ISBN: 978-1-5386-3464-6. DOI: 10.1109/FOCS.2017.32.

[52] Giorgio Gambosi, Alberto Postiglione, and Maurizio Talamo. "Algorithms for the Relaxed Online Bin-Packing Model." In: *SIAM Journal on Computing* 30.5 (2000), pp. 1532–1551. DOI: 10.1137/S0097539799180408.

[53] Michael R. Garey and David S. Johnson. "" Strong " NP-Completeness Results: Motivation, Examples, and Implications." In: *Journal of the ACM* 25.3 (July 1978), pp. 499–508. ISSN: 0004-5411. DOI: 10.1145/322077.322090.

[54] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN: 0-7167-1044-7.

[55] Michael R. Garey and David S. Johnson. "Approximation Algorithms for Bin Packing Problems: A Survey." In: *Analysis and Design of Algorithms in Combinatorial Optimization*. Ed. by G. Ausiello and M. Lucertini. Vienna: Springer Vienna, 1981, pp. 147–172. ISBN: 978-3-7091-2748-3. DOI: 10.1007/978-3-7091-2748-3_8.

[56] Michael R. Garey, Ronald L. Graham, and Jeffrey D. Ullman. "Worst-Case Analysis of Memory Allocation Algorithms." In: *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*. ACM, 1972, pp. 143–150.

[57] Michael R. Garey, Ronald L. Graham, David S. Johnson, and Andrew C. Yao. "Resource constrained scheduling as generalized bin packing." In: *Journal of Combinatorial Theory, Series A* 21.3 (1976), pp. 257 –298. ISSN: 0097-3165. DOI: 10.1016/0097-3165(76)90001-7.

[58] Paul C. Gilmore and Ralph E. Gomory. "A Linear Programming Approach to the Cutting-Stock Problem." In: *Operations Research* 9.6 (1961), pp. 849–859. DOI: 10.1287/opre.9.6.849.

[59] Gregory Gutin, Tommy R. Jensen, and Anders Yeo. "Batched bin packing." In: *Discrete Optimization* 2.1 (2005), pp. 71–82. DOI: 10.1016/j.disopt.2004.11.001.

[60] Xin Han, Deshi Ye, and Yong Zhou. "A note on online hypercube packing." In: *Central European Journal of Operations Research* 18.2 (June 2010), pp. 221–239. DOI: 10.1007/s10100-009-0109-z.

[61] Xin Han, Francis Y. L. Chin, Hing-Fung Ting, Guochuan Zhang, and Yong Zhang. "A new upper bound 2.5545 on 2D Online Bin Packing." In: *ACM Transactions on Algorithms* 7.4 (2011), 50:1–50:18. DOI: 10.1145/2000807.2000818.

[62] Rolf Harren. "Approximation algorithms for orthogonal packing problems for hypercubes." In: *Theoretical Computer Science* 410.44 (2009), pp. 4504–4532. DOI: 10.1016/j.tcs.2009.07.030.

[63] Rolf Harren and Rob van Stee. "Absolute approximation ratios for packing rectangles into bins." In: *Journal of Scheduling* 15.1 (2012), pp. 63–75. DOI: 10.1007/s10951-009-0110-3.

[64] Rolf Harren, Klaus Jansen, Lars Prädel, Ulrich M. Schwarz, and Rob van Stee. "Two for One: Tight Approximation of 2D Bin Packing." In: *International Journal of Foundations of Computer Science* 24.8 (Dec. 2013), pp. 1299–1328. DOI: 10.1142/S0129054113500354.

[65] Sandy Heydrich and Rob van Stee. "Beating the Harmonic Lower Bound for Online Bin Packing." In: *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*. Ed. by Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi. Vol. 55. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 41:1–41:14. ISBN: 978-3-95977-013-2. DOI: 10.4230/LIPIcs.ICALP.2016.41.

[66] Sandy Heydrich and Andreas Wiese. "Faster approximation schemes for the two-dimensional knapsack problem." In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*. Ed. by Philip N. Klein. SIAM, 2017, pp. 79–98. ISBN: 978-1-61197-478-2. DOI: 10.1137/1.9781611974782.6.

[67] Rebecca Hoberg and Thomas Rothvoß. "A Logarithmic Additive Integrality Gap for Bin Packing." In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*. 2017, pp. 2616–2625. DOI: 10.1137/1.9781611974782.172.

[68] Wiebke Höhn, Julián Mestre, and Andreas Wiese. "How Unsplittable-Flow-Covering Helps Scheduling with Job-Dependent Cost Functions." In: *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*. Ed. by Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias. Vol. 8572. Lecture Notes in Computer Science. Springer, 2014, pp. 625–636. ISBN: 978-3-662-43947-0. DOI: 10.1007/978-3-662-43948-7_52.

[69] Oscar H. Ibarra and Chul E. Kim. "Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems." In: *Journal of the ACM* 22.4 (1975), pp. 463–468. DOI: 10.1145/321906.321909.

[70] Klaus Jansen and Lars Prädel. "New Approximability Results for Two-Dimensional Bin Packing." In: *Algorithmica* 74.1 (2016), pp. 208–269. DOI: 10.1007/s00453-014-9943-z.

[71] Klaus Jansen and Roberto Solis-Oba. "New Approximability Results for 2-Dimensional Packing Problems." In: *Mathematical Foundations of Computer Science 2007, 32nd International Symposium, MFCS 2007, Ceský Krumlov, Czech Republic, August 26-31, 2007, Proceedings*. Ed. by Ludek Kucera and Antonín Kucera. Vol. 4708. Lecture Notes in Computer Science. Springer, 2007, pp. 103–114. ISBN: 978-3-540-74455-9. DOI: 10.1007/978-3-540-74456-6_11.

[72] Klaus Jansen and Roberto Solis-Oba. "Packing Squares with Profits." In: *SIAM Journal on Discrete Mathematics* 26.1 (2012), pp. 263–279. DOI: 10.1137/080717110.

[73] Klaus Jansen and Rob van Stee. "On strip packing With rotations." In: *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*. Ed. by Harold N. Gabow and Ronald Fagin. ACM, 2005, pp. 755–761. ISBN: 1-58113-960-8. DOI: 10.1145/1060590.1060702.

[74] Klaus Jansen and Guochuan Zhang. "Maximizing the Number of Packed Rectangles." In: *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*. Ed. by Torben Hagerup and Jyrki Katajainen. Vol. 3111. Lecture Notes in Computer Science. Springer, 2004, pp. 362–371. ISBN: 3-540-22339-8. DOI: 10.1007/978-3-540-27810-8_31.

[75] Klaus Jansen and Guochuan Zhang. "Maximizing the Total Profit of Rectangles Packed into a Rectangle." In: *Algorithmica* 47.3 (2007), pp. 323–342. DOI: 10.1007/s00453-006-0194-5.

[76] David S. Johnson. "Fast algorithms for bin packing." In: *Journal of Computer and System Sciences* 8.3 (1974), pp. 272 –314. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(74)80026-7. URL: http://www.sciencedirect.com/science/article/pii/S0022000074800267.

[77] David S. Johnson, Alan Demers, Jeffrey D. Ullman, Michael R. Garey, and Ronald L. Graham. "Worst-case performance bounds for simple one-dimensional packing algorithms." In: *SIAM Journal on Computing* 3.4 (1974), pp. 299–325.

[78] L. V. Kantorovich. "Mathematical Methods of Organizing and Planning Production." In: *Management Science* 6.4 (1960), pp. 366–422. DOI: 10.1287/mnsc.6.4.366.

[79] Leonid V. Kantorovich. *Mathematical methods of organizing and planning production*. Leningrad State University. 1939.

[80] Narendra Karmarkar and Richard M. Karp. "An Efficient Approximation Scheme for the One-Dimensional Bin-Packing Problem." In: *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*. IEEE Computer Society, 1982, pp. 312–320. DOI: 10.1109/SFCS.1982.61.

[81] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004. ISBN: 978-3-540-40286-2.

[82] Claire Kenyon and Eric Rémila. "A near-optimal solution to a two-dimensional cutting stock problem." In: *Mathematics of Operations Research* 25.4 (2000), pp. 645–656. DOI: 10.1287/moor.25.4.645.12118.

[83] Yoshiharu Kohayakawa, Flávio Keidi Miyazawa, Prabhakar Raghavan, and Yoshiko Wakabayashi. "Multidimensional Cube Packing." In: *Algorithmica* 40.3 (2004), pp. 173–187. DOI: 10.1007/s00453-004-1102-5.

[84] Yoshiharu Kohayakawa, Flávio Keidi Miyazawa, and Yoshiko Wakabayashi. "A tight lower bound for an online hypercube packing problem and bounds for prices of anarchy of a related game." In: *LATIN 2018: Theoretical Informatics, 13th Latin American Symposium, Proceedings*. 2018, To appear.

[85] Ariel Kulik and Hadas Shachnai. "There is no EPTAS for two-dimensional knapsack." In: *Information Processing Letters* 110.16 (2010), pp. 707–710. DOI: 10.1016/j.ipl.2010.05.031.

[86] C. C. Lee and D. T. Lee. "A Simple On-Line Bin-Packing Algorithm." In: *Journal of the ACM* 32.3 (1985), pp. 562–572. DOI: 10.1145/3828.3833.

[87] Joseph Y.-T. Leung, Tommy W. Tam, C. S. Wong, Gilbert H. Young, and Francis Y. L. Chin. "Packing Squares into a Square." In: *Journal of Parallel and Distributed Computing* 10.3 (1990), pp. 271–275. DOI: 10.1016/0743-7315(90)90019-L.

[88] Frank M. Liang. "A Lower Bound for On-Line Bin Packing." In: *Information Processing Letters* 10.2 (Mar. 1980), pp. 76–79. DOI: 10.1016/S0020-0190(80)90077-0.

[89] László Lovász and Michael D. Plummer. *Matching Theory*. Akadémiai Kiadó - North Holland, Budapest, 1986.

[90] Michael J. Magazine and Maw-Sheng Chern. "A Note on Approximation Schemes for Multidimensional Knapsack Problems." In: *Mathematics of Operations Research* 9.2 (1984), pp. 244–247. DOI: 10.1287/moor.9.2.244.

[91] Weizhen Mao. "Tight Worst-Case Performance Bounds for Next-k-Fit Bin Packing." In: *SIAM Journal on Computing* 22.1 (1993), pp. 46–56. DOI: 10.1137/0222004.

[92] G. B. Mathews. "On the partition of numbers." In: *Proceedings of the London Mathematical Society* 28 (1897), pp. 486–490.

[93] Flávio Keidi Miyazawa and Yoshiko Wakabayashi. "Packing Problems with Orthogonal Rotations." In: *LATIN 2004: Theoretical Informatics, 6th Latin American Symposium, Buenos Aires, Argentina, April 5-8, 2004, Proceedings*. Ed. by Martin Farach-Colton. Vol. 2976. Lecture Notes in Computer Science. Springer, 2004, pp. 359–368. ISBN: 3-540-21258-2. DOI: 10.1007/978-3-540-24698-5_40.

[94] Giorgi Nadiradze and Andreas Wiese. "On approximating strip packing with a better ratio than 3/2." In: *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*. Ed. by Robert Krauthgamer. SIAM, 2016, pp. 1491–1510. ISBN: 978-1-61197-433-1. DOI: 10.1137/1.9781611974331.ch102.

[95] Prakash V. Ramanan, Donna J. Brown, C. C. Lee, and D. T. Lee. "On-Line Bin Packing in Linear Time." In: *Journal of Algorithms* 10.3 (1989), pp. 305–326. DOI: 10.1016/0196-6774(89)90031-X.

[96] Rasterweq software. *Smart2DCutting*. http://www.rasterweq.com/specifications.htm.

[97] Thomas Rothvoß. "Approximating Bin Packing within O(log OPT * Log Log OPT) Bins." In: *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*. IEEE Computer Society, 2013, pp. 20–29. ISBN: 978-0-7695-5135-7. DOI: 10.1109/FOCS.2013.11.

[98] Steven S. Seiden. "On the online bin packing problem." In: *Journal of the ACM* 49.5 (2002), pp. 640–671. DOI: 10.1145/585265.585269.

[99] Steven S. Seiden and Rob van Stee. "New Bounds for Multidimensional Packing." In: *Algorithmica* 36.3 (2003), pp. 261–293. DOI: 10.1007/s00453-003-1016-7.

[100] David B Shmoys and Éva Tardos. "An approximation algorithm for the generalized assignment problem." In: *Mathematical Programming* 62.1-3 (1993), pp. 461–474. DOI: 10.1007/BF01585178.

[101] David Simchi-Levi. "New worst-case results for the bin-packing problem." In: *Naval Research Logistics* 41.4 (1994), p. 579.

[102] Daniel D. Sleator and Robert E. Tarjan. "Amortized Efficiency of List Update and Paging Rules." In: *Communications of the ACM* 28.2 (1985), pp. 202–208. DOI: 10.1145/2786.2793.

[103] TMachines. *CutLogic2D*. https://www.tmachines.com/cutlogic-2d/.

[104] Jeffrey D. Ullman. *The performance of a memory allocation algorithm*. Tech. rep. 100. Princeton University, Princeton, NJ, 1971.

[105] Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001. ISBN: 978-3-540-65367-7.

[106] André van Vliet. "An Improved Lower Bound for On-Line Bin Packing Algorithms." In: *Information Processing Letters* 43.5 (1992), pp. 277–284. DOI: 10.1016/0020-0190(92)90223-I.

[107] André van Vliet. "Lower and upper bounds for online bin packing and scheduling heuristics." PhD thesis. Rotterdam, The Netherlands: Erasmus University, 1995.

[108] Andrew C. Yao. "New Algorithms for Bin Packing." In: *Journal of the ACM* 27.2 (1980), pp. 207–227. DOI: 10.1145/322186.322187.

[109] Guochuan Zhang. "A 3-approximation algorithm for two-dimensional bin packing." In: *Operations Research Letters* 33.2 (2005), pp. 121–126. DOI: 10.1016/j.orl.2004.04.004.