CHRISTIAN-ALBRECHTS UNIVERSITÄT ZU KIEL

DISSERTATION

On Algorithms for Multidimensional Packing Problems and on the Complexity of higher dimensional Knapsack

Author:
Kilian GRAGE

Betreuer:
Professor Dr. Klaus JANSEN

Dissertation zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

Arbeitsgruppe Algorithmen und Komplexität
Technische Fakultät

Eingereicht im April 2024

- 1. Gutachter: Prof. Dr. Klaus Jansen
- 2. Gutachter: Prof. Dr. Nicole Megow

Datum der Disputation: 16. Oktober 2024

CHRISTIAN-ALBRECHTS UNIVERSITÄT ZU KIEL

Abstract

Technische Fakultät

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

On Algorithms for Multidimensional Packing Problems and on the Complexity of higher dimensional Knapsack

by Kilian GRAGE

This thesis consists of two parts. In the first half we consider a range of so called online problems. In an online setting we are not given all informations of our input instance immediately. Instead we receive informations over time and algorithms have to handle not only the given instance at a specific time, but also the lack of information and an unknown future.

We consider a slightly relaxed setting where the solution we generate is not fixed but can be altered. We do not allow arbitrary repacking. We measure objects by some size criteria and allow repacking based on the total size of arrived or departed objects.

For this setting we introduce and analyse a framework that is based on the very simple idea of combining known offline results with flexible and simple online algorithms. The resulting algorithm of our framework is then easily described: As we receive more information of our input data, we apply an online algorithm, that is suited to handle the new influx of data and is capable of maintaining a certain solution quality despite not knowing what happens in the future.

At specific points in time, when solution quality starts to deteriorate, we then repack the solution. For this repacking we use offline algorithms with good approximation ratios to calculate an efficient solution that we continue working with. For the next changes of the instance we then continue again with the online algorithm and keep alternating between both online and offline algorithm. Surprisingly this achieves a solution quality close to quality of the offline algorithm.

The second half of this thesis concerns itself with the multidimensional Knapsack problem and its connection to the $(\max, +)$-convolution problem. We show that a conditional lower bound from the one-dimensional case can be generalized to higher dimensions. This implies that if and only if $(\max, +)$-convolution in higher dimension can not be solved in sub-quadratic time, that even knapsack can not be solved in time that is sub-quadratic in the number of knapsack capacities.

We complement these results with algorithms that also abuse the connection between Knapsack and Convolution. We give a new algorithm that solves higher dimensional knapsack using one-dimensional convolution and carry over these results in order solve general Integer Linear Programs as well. Finally we also give an algorithm to solve Malleable Job Scheduling. This last algorithm is also based on the idea of using Knapsack as a sub-problem and uses Convolution in order to solve the Knapsack-instance.

Zusammenfassung

On Algorithms for Multidimensional Packing Problems and on the Complexity of higher dimensional Knapsack

by Kilian GRAGE

Diese Thesis besteht aus zwei Teilen. In der ersten Hälfte betrachten wir eine Auswahl von sogenannten Online Problemen. In der Online Version eines Problems sind uns nicht alle Informationen der Eingabe von Anfang an gegeben, sondern wir erhalten Informationen im Laufe der Zeit. Daher müssen Algorithmen nicht nur eine zu einem Zeitpunkt gegebene Instanz lösen, sondern müssen auch mit dem Mangel an Information und einer unbekannten Zukunft zurecht kommen.

Wir schauen uns eine Variante an, in der unsere berechnete Lösung nicht in Stein gemeißelt ist und wir diese auch zu einem späteren Zeitpunkt verändern können. Wir beschränken uns allerdings und erlauben nicht beliebige Änderungen. Statt dessen messen wir die Veränderungen der Instanz anhand eines Kriteriums (zum Beispiel die Größe eines Objektes) und beschränken Veränderungen basierend auf der gesamten Größe von neuen oder entfernten Objekten.

Für dieses Setting stellen wir ein neues Framework vor und analysieren dieses. Das Framework basiert auf der einfachen Idee zwei Algoritmen, einen bereits vorhanden Algorithmus aus der Offline-Welt und einen einfachen und flexiblen Online-Algorithmus zu kombinieren. Das daraus entstehende Framework lässt sich leicht erklären: Im Verlaufe der Zeit werden neue Informationen und Objekte durch den Online-Algorithmus verarbeitet. Dieser ist prinzipiell gut dafür geeignet, die neuen Objekte zu verwalten und kann damit eine akzeptable Lösungsqualität garantieren.

Zu gewissen Zeitpunkten, wenn die Lösungsqualität sich langsam verschlechtert, wollen wir dann eine neue Lösung berechnen. Dafür benutzen wir dann einen guten Offline-Algorithmus mit einer guten Approximationsgüte um eine effiziente Lösung zu berechnen, mit der wir dann weiterarbeiten können. Für die darauffolgenden Änderungen, benutzen wir dann wieder den Online-Algorithmen und alternieren weiter auf diese Art und Weise zwischen den beiden Algorithmen. Überraschenderweise ist die Qualität der Lösung nur geringfügig schlechter, als die des Offline-Algorithmus.

Die zweite Hälfte dieser Dissertation geht um das mehrdimensionale Rucksack-Problem und seiner Verbindung zum $(\max, +)$-convolution Problem. Wir zeigen, dass eine bedingte untere Schranke vom eindimensionalen Rucksack Problem sich erweitern lässt in höhere Dimension. Die Konsequenz ist dann: Sofern $(\max, +)$-convolution in höheren Dimension nicht in einer sub-quadratischen Laufzeit gelöst werden kann, so kann auch das Rucksackproblem in höheren Dimension nicht mit einer sub-quadratischen Laufzeit gelöst werden.

Wir komplementieren diese unteren Schranken mit Algorithmen, die auch die Verbindung zwischen Rucksack und Convolution nutzen. Wir stellen einen Rucksack Algorithmus vor, welcher als Teilproblem ein-dimensionale Convolution lösen muss. Diese Techniken nutzen wir weiter um auch allgemeine Integer Linear Programs auch zu lösen. Final stellen wir auch einen Algorithmus für das Malleable Job Scheduling Problem vor. Dieser Algorithmus basiert auch auf der Idee eine Rucksack-Instanz zu lösen und nutzt dafür das Convolution-Problem.

Acknowledgements

I would like to wholeheartedly thank my supervisor Klaus Jansen for offering me the possibility to join his group and making all this possible for me. Furthermore I want to thank all my colleagues from Kiel University, co-authors and collaborators Sebastian Berndt, Hauke Brinkop, Max Deppert, Valentin Dreismann, Leah Epstein, Ute Iaquinto, Lukas Johannsen, Kim-Manuel Klein, Ingmar Knopf, Maria Kosche, Felix and Kati Land, Alexandra Lassota, Asaf Levin, Marten Maack, Parvaneh Massouleh, Felix Ohnesorge, Malin Rau and Lars Rohwedder.

Further I want to thank my family, in particular my mother Stephanie, my stepfather Hans for always being there for me. Last but not least I want to thank all my friends who helped and supported me during this time.

Dedicated to my mother Stephanie and my brother Tarik.

Contents

Chapter 1

Introduction

Packing problems are a range of very interesting problems that stem from simple real world examples. Imagine we want to fly into holidays and we need to pack our luggage for the flight. We probably have a lot of items that we would like to bring, but realistically we are going to be limited by multiple factors.

First of all we are probably limited by the number of travel suitcases available to us. These can be further limited if the flight does only allow up to a specific number of suitcases. Furthermore each suitcase might be further restricted. Maybe there is a weight limit for each individual suitcase or maybe certain items are only allowed up to a certain quantity. All these factors make for quite a difficult problem - in theory.

In reality people apply individual strategies not thinking too much about the efficiency of their packing. Some people may pack similar things together, to be able to find them easier together later. Some people may actually prefer to prepare the things outside of the suitcase and then pack things and see how they fit well together to save space. Finally there is also a group of people that just throw their stuff in and pray that at the end everything fits. While these strategies work for each individual and are sufficient for a lax context such as holidays, one can still ask at the end of the day: "Could I have done better?". And the answer to that question is most definitely yes.

In Computer Science we aim to solve problems and usually we aim to solve them optimally. However some problems have proven to be quite difficult to solve in a reasonable time. For these NP-hard problems up to this date there have been no optimal polynomial time algorithms. Therefore there are two approaches that one can follow:

Either we relax the problem and try to find an approximate solution. We still want to guarantee that our solution - in regards to some measurable optimisation criteria - is close to the optimum, but the main goal is to forgo an optimal solution to achieve a polynomial time algorithm.

The other approach is to make further restrictions to the problems and make additional assumptions about relevant parameters of the problem. For example in the setting of our holiday luggage it might be a reasonable assumption that our items are not too big. We will probably not have any items that will fill out a whole suitcase by itself and even if we actually do, we might just put it into a separate suitcase and focus on the rest of the luggage. Assumptions like that - in this case about the size parameter being small - can enable efficient algorithmic strategies that help us solve problems in a more reasonable time.

The packing problems we consider in this thesis are all NP-hard and therefore we will need to make due with either of these alternative strategies to achieve polynomial time. Generally we will consider two different kind of problems based on whether we aim to maximize or minimize their optimisation criteria.

In a minimization problem our problem is usually linked to some kind of cost or something similar we want to keep small. In our holiday luggage setting it could be the number of suitcases we bring. It seems very likely that if we bring more suitcases to the flight, we will also be charged more at the end. Consider a very simplistic model of this problem: Every item is simply defined by some size value and that we are given an arbitrary number of suitcases that all have the same capacity, which is also given by a size value. This problem in its core is known as Bin Packing where we aim to minimize the number of suit cases or Bins necessary to pack all our given items.

For the maximization setting contrary to the minimization problem we aim to maximize some value. Bin Packing is not an ideal model for our holiday luggage, cause our goal might not be to bring all of our items. Instead we are more likely to have limited space and try to fill it with as much value as we can. This in its core is known as Knapsack Problem, where we are given a set of items and each item has a size and a profit value. Our knapsack - or suitcase - also has a size capacity. The goal for this problem is then to fill the knapsack with items, that do not exceed the size limit, but have a high profit value.

Bin Packing and Knapsack are probably the most simple variants of packing problems, but can be generalized to even harder problems. The first natural generalization is adding more size parameters. Instead of a singular size value we identify sizes with vectors then - both for items and knapsacks/bins. This is the higher dimensional variant of each respective problem.

One could also add a geometric layer to this problem. In none of the aforementioned models we actually bothered finding a solution for how to actually pack the suitcase. We could therefore model items as 2-dimensional rectangles or 3-dimensional cuboids that need to be fitted into a knapsack/bin that is also represented by the same structure. This immediately adds another completely different and difficult layer to the problem: Not only does the choice of items for each container remain relevant, but also how these are packed.

These packing problems among others will be considered in the following. We will also look at some problems that do not immediately qualify as packing problems, but where the choice of items is the main question. Some graph problems such as Vertex Cover or Maximum Independent Set will fall under this category.

1.1 Structure of the Thesis

In the first two chapters we will look at a wide range of problems in the online setting. In an online setting we are not given the full instance from the beginning. Instead items arrive over time and we need to pack them as they come and - potentially even - go. We allow for repacking the solution but limit ourselves to not allow arbitrary repacking based on amortized migration. This in particular means that we measure the size of arriving and departed items and only allow repacking based on the total size.

For this setting we introduce a framework that uses online algorithm to handle changes on the instance, but an offline algorithm for the repackings. In chapter 2 we discuss how the framework can be applied to minimization problems and in chapter 3 we extend the results to maximization problems. Both of these chapters are based on the publications [14, 16].

In the second half we specialise in particular on the Knapsack Problem in higher dimension. We generalize in chapter 4 a conditional lower bound and a connection to the (max, +)-convolution problem that was proven by Cygan et al. [36] for the one dimensional case. As an interesting sub-result we achieve an algorithm for multi-dimensional knapsack using techniques that Bringmann used in the one-dimensional scenario.

We also give a parameterized algorithm, generalized from Axiotis and Tzamos [2] solving d-dimensional Knapsack using convolution as a sub-problem. Note that we not only generalize the result from Axiotizs and Tzamos but we also show that we can skip one step of their algorithm. We also carry these results over to the ILP world by applying a reduction to handle negative values. Compared to the long standing result of Eisenbrand and Weismantel our algorithm achieves a slightly better dependency on the dimension/number of constraints while being slightly worse for larger absolute entries in the constraint matrix. The contents of chapter 4 are based on [51].

Finally in chapter 5 we give an algorithm that solves a scheduling problem with malleable monotone jobs. The algorithm at its core solves a Knapsack problem, which in turn is solved via techniques based on convolution. The knapsack problem uses ideas from Mounié et al. [76] as well as the concept of compression from Jansen and Land [59]. The topics discussed in this chapter were published in [52].

Chapter 2

Robust Online Algorithms for Minimization Problems

Online algorithms that allow a small amount of migration or recourse have been intensively studied in the last years. They are essential in the design of competitive algorithms for dynamic problems, where objects can also depart from the instance. In this part, we give a general framework to obtain so called robust online algorithms for a variety of dynamic problems: these online algorithms achieve an asymptotic competitive ratio of $\gamma + \epsilon$ with migration $O(1/\epsilon)$, where γ is the best known offline asymptotic approximation ratio. For our framework, we require only two ingredients: (i) the existence of an online algorithm for the static case (without departures) that provides a provably good solution compared to the total volume of the instance and (ii) that the optimal solution always exceeds this total volume. If these criteria are met, we can complement the online algorithm with any offline algorithm.

While these criteria are naturally fulfilled by many dynamic problems, they are especially suited for packing problems. In order to show the usefulness of our approach in this area, we improve upon the best known robust algorithms for the dynamic versions of generalizations of Strip Packing and Bin Packing, including the first robust algorithms for general d-dimensional Bin Packing and Vector Packing.

2.1 Introduction

Online algorithms are a very natural way to deal with uncertain inputs and the need for a sequence of good solutions throughout the evolution of an instance. For a surprisingly large number of problems, one can obtain online algorithms that produce solutions within a constant ratio of the optimal solutions. The worst-case ratio between an optimal solution and a solution produced by the online algorithm is called the *competitiveness* of the algorithm. For most algorithms with constant competitiveness, the algorithms rely heavily on the fact that the instances evolve in a *monotone* way: every object that becomes part of the instance will stay part of the instance forever. This monotonicity property is often not present in real-world applications, where the objects might be removed from the instance later on. These objects might be used at a different place, they might expire, or they are no longer relevant. Hence, in order to still give a performance guarantee, the online algorithms need to be able to modify parts of an existing solutions. Clearly, such a modification might be costly and should thus be minimized. If no such boundary on the modification is given, one could easily use an offline approximation algorithm. A natural way to measure the amount of modification needed is the *migration factor*: it compares the total size of modified objects with the size of the newly inserted or removed object. An

algorithm with a bounded migration factor roughly translates to the fact that the insertion or departure of a small object can only lead to small changes in the structure of the current solution. On the other hand, if a large (and thus impactful) object is inserted or removed, we are allowed to modify a larger part of the solution.

It is easy to see that online algorithms can in general not achieve the same solution quality as offline algorithms. The question how much these two values differ is one of the central questions in the field of online and approximation algorithms. Clearly, an online algorithm that is allowed a certain amount of migration is able to gap between these extremes. If the amount of migration needed directly corresponds to the improvement of the solution guarantee, we call such an algorithm *robust*. As an example, consider the well-studied Bin Packing problem. No online algorithm (without migration) can achieve a competitiveness smaller than $3/2$ [84], while an offline algorithm with asymptotic approximation ratio of $1+\epsilon$ can be obtained in polynomial time [68]. A robust online algorithm would then have competitiveness $1+\epsilon$ and migration factor $f(1/\epsilon)$ for some function. In other words, we only need to increase the migration if we want to obtain a better solution. Such robust online algorithms thus have a continuous behavior between the performance of the best offline algorithm ($\epsilon \to 0$) and the performance of the best online algorithm ($\epsilon \to \infty$).

In this section, we describe a general framework to construct robust online algorithms for dynamic online problems. The only ingredients needed for our framework are (i) a good offline approximation algorithm and a (ii) suitably designed online algorithm for the *static* case. Our framework is applicable to many problems, especially in the field of geometric packing problems. To show the versatility of our approach, we use it to improve upon existing robust algorithms and to construct new robust algorithms.

2.1.1 Related Works

The migration factor model was introduced by Sanders, Sivadasan and Skutella [78]. They studied the classical problem of minimizing the makespan on parallel identical machines and made use of sensitivity results of integer programming to obtain a robust $1+\epsilon$-competitive algorithm. This work spanned a lot of follow up works for a wide range of different problems:

- Skutella and Verschae [79] also studied the same problem and were able to obtain a robust $1+\epsilon$-competitive algorithm for the dynamic case, where jobs may depart. They also studied the dual version of makespan minimization problem, where the minimum load should be maximized. This problem (called Machine Covering or Santa Claus) also admits a robust $1+\epsilon$-competitive algorithm even in the dynamic case. Skutella and Verschae also proved that for all $\epsilon > 0$ there is no online algorithm for the Machine Covering problem that achieves competitive ratio $20/19-\epsilon$ with worst-case migration $f(1/\epsilon)$ for any function f. Gálvez *et al.* [48] gave two simple competitive robust algorithms (one that is $1.7+\epsilon$-competitive and one that is $4/3+\epsilon$-competitive) with polynomial migration factor for the static version of the problem of makespan minimization.

- For the Bin Packing problem, Epstein and Levin [42] gave the first robust $1+\epsilon$-competitive algorithm for Bin Packing based on the same sensitivity results for integer programming. Jansen and Klein [62] designed new techniques in order to obtain a migration factor polynomial in $1/\epsilon$. These techniques were improved by Berndt *et al.* [12] to also handle the dynamic version of the Bin

Packing problem where items can also depart. In all of these works, the migration factor is worst-case and can thus not be saved up for later use. In contrast, Feldkord *et al.* [47] presented a simple robust $1+\epsilon$-competitive algorithm with amortized migration factor that also works for the dynamic case. Berndt *et al.* also showed that a worst-case migration factor of $\Omega(1/\epsilon)$ is needed [12]. This lower bound was shown to also hold for amortized migration by Feldkord *et al.* [47]. Feldkord *et al.* also studied a problem variant, where the migration needed for an item does not correspond to its size.

- For the makespan problem, where jobs can be scheduled preemptively (i. e. they can be split, but parts are not allowed to run simultaneously), Epstein and Levin [43] designed an optimal online algorithm with migration factor $1 - 1/m$, where m is the number of machines. Again, the migration factor used here is worst-case. They also showed that exact algorithms for the makespan minimization problem on uniform machines and for identical machines in the restricted assignment have worst-case migration factor at least $\Omega(m)$.

- Berndt *et al.* [13] studied the dual problem of Bin Packing called Bin Covering. They analyzed this problem both for worst-case migration and amortized migration and for the static and the dynamic case to obtain lower bounds and matching algorithmic results (up to an additional additive term of ϵ).

The offline variants of the geometric packing problems have also been studied intensively. See e. g. the survey of Christensen *et al.* [32] and the references therein.

2.1.2 Our Results

We present a very general framework that allows to construct robust online algorithms that have optimal amortized migration factor of $O(1/\epsilon)$ and achieve the same competitive ratio as the best known offline approximation algorithm (up to an additive error of ϵ). This framework is suitable for many different optimization problems, especially geometric packing problems. The algorithms created by our framework can all deal with the dynamic versions of these problems, where items can also depart from the instance. To present a general framework, we introduce the notion of *flexible online algorithms* and show that such algorithms can be combined with offline approximation algorithms under certain circumstances. We then show the versatility of our approach by looking at several well-studied problems including generalizations of Strip Packing and Bin Packing. We give robust algorithms for the multi-dimensional variants of these problems, where we can also handle both the departure and the rotation of items. This improves and generalizes several known results and gives the first robust algorithms for all of the other problems. Additionally, our compact and clean framework gives much easier algorithms compared with the previously known algorithms.

2.2 Online Algorithms for Dynamic Problems: A Framework

2.2.1 Preliminaries

In this section we will discuss minimization problems. We remark that these techniques also work for maximization problems but we will discuss these in the next section.

Definition 2.2.1. Let ITEMS $\subseteq \{0,1\}^*$ be a prefix-free set describing some items. A *minimization problem* $\Pi = (\mathcal{I}, \text{SOL}, \text{COSTS})$ consist of a set of *instances* $\mathcal{I} \subseteq$ ITEMS, a mapping SOL that maps an instance $I \in \mathcal{I}$ to a non-empty set of *feasible solution* $\text{SOL}(I)$, and a mapping COSTS that maps a solution $S \in \text{SOL}(I)$ to its *costs* $\text{COSTS}(S) \in \mathbb{Q}_{\geq 0}$. A solution $S^* \in \text{SOL}(I)$ is called *optimal*, if $\text{COSTS}(S^*) \leq \text{COSTS}(S)$ for all $S \in \text{SOL}(I)$. We denote the cost of any optimal solution S^* by $\text{OPT}(I) := \text{COSTS}(S^*)$.

For the sake of simplicity, we also sometimes treat Π and $\mathcal{I}$ interchangeably and also write $I \in \Pi$ to denote that I is an instance.

We make the following two natural assumptions that hold for all problems considered in this section:

Assumption 1.

1. *For every instance I, every instance $I' \subseteq I$, and every solution $S \in \text{SOL}(I)$, the solution $S \restriction_{I'}$ induced by the items in I' is also feasible, i.e. $S \restriction_{I'} \in \text{SOL}(I')$. Hence, removing some items from a feasible solution results in a feasible solution of the remaining items.*
2. *For every instance I, and every instance $I' \subseteq I$, we have $\text{OPT}(I') \leq \text{OPT}(I)$. Hence, removing items from an instance can only decrease the optimum.*

In the classical *offline version* of $\Pi = (\mathcal{I}, \text{SOL}, \text{COSTS})$, we are given the complete instance $I \in \mathcal{I}$ all in once. An *α-approximation* for Π is a polynomial-time algorithm A such that for all instances $I \in \mathcal{I}$, we have $\text{A}(I) \leq \alpha \, \text{OPT}(I) + c$ for some constant c not depending on the instance. Here, $\text{A}(I)$ is the value of the solution produced by A.

In the *online version* of a minimization problem $\Pi = (\mathcal{I}, \text{SOL}, \text{COSTS})$, an *online instance* I is a sequence of instances $I_1, I_2, \ldots, I_{|I|} \in \mathcal{I}$ such that $|I_t \triangle I_{t+1}| = 1$ for all $t = 1, \ldots, |I| - 1$, where $\triangle$ is the symmetric difference of two sets. This means that we *insert* a new item i^* (if $I_{t+1} \setminus I_t = \{i^*\}$) or an item i^* *departs* (if $I_t \setminus I_{t+1} = \{i^*\}$). The set of online instances of Π is denoted as $\mathcal{I}^{\text{on}}$. An *online algorithm* A maintains a sequence of solutions $S_1, \ldots, S_t$, where $S_t \in \text{SOL}(I_t)$ and furthermore, $S_{t+1}(i) = S_t(i)$ for all $i \in I_{t+1} \cap I_t$. In the *static online version*, we only have insertions and thus $I_t \subsetneq I_{t+1}$. A non-static problem is called *dynamic*. We say that an online algorithm A producing such a sequence of solutions $S_1, S_2, \ldots$ is *β-competitive*, if

$$\text{COSTS}(S_t) \leq \beta \cdot \text{OPT}(I_t) + c$$

for all $t = 1, \ldots, |I|$ and some constant c not depending on the instance. This notion of competitiveness is sometimes also called *asymptotic competitiveness* in contrast to the notion of *absolute competitiveness*, where no additional additive term c is allowed. Whenever we talk about competitiveness, this is with regard to the notion of asymptotic competitiveness.

Migration Achieving bounded competitiveness is usually impossible for dynamic problems, even for very simple problems such as Bin Packing where every item has the same size. This is simply due to the fact that an online algorithm is not allowed to change the placement of already placed items. This very strict restriction thus comes with a high cost with regard to the competitiveness. As many real-world applications are not only static, but allow the departure of items, one must thus me more flexible. We will thus allow a small amount of repacking to be able to handle dynamic problems. The model of repacking we use is called the *amortized migration factor model*. In this setting, every item $i \in$ ITEMS comes with a *size* $v_i \in \mathbb{Q}_{\geq 0}$. Usually

this size will be the space needed for an item, like it's total area or volume or some identical criteria like the side length of a hypercube. For a set of items $I \subseteq \text{ITEMS}$, we denote by $\text{VOL}(I) = \sum_{i \in I} v_i$ the complete volume of I. If $I \in \mathcal{I}^{\text{on}}$ is an online instance and $\vec{S} = (S_1, \ldots, S_{|I|})$ is a sequence of solutions with $S_j \in \text{SOL}(I_j)$, the *migrated items* $M_t(\vec{S})$ at time t are defined as $M_t(\vec{S}) = \{i \in I_{t-1} \cap I_t \mid S_{t-1}(i) \neq S_t(i)\}$. The total *migration* $\mu(\vec{S}, t)$ used until time t is defined as $\mu(\vec{S}, t) := \sum_{j=1}^{t} \sum_{i \in M_j(\vec{S})} v_i$, i. e. as the sum of the sizes of the migrated items. Inserting an item i builds up a migration potential of v_i and migrating i has cost v_i. More formally, for $t = 1, \ldots, |I|$, let $A_t = \bigcup_{j \in \{0,1,\ldots,t-1\}} (I_{j+1} \setminus I_j)$ be the set of items that were inserted until time t and $D_t = \bigcup_{j \in \{0,1,\ldots,t-1\}} (I_j \setminus I_{j+1})$ be the set of items that departed until time t. Let A be an online algorithm that is allowed to migrate items. We say that A has *migration factor* β, if

$$\mu(\vec{S}, t) \leq \beta[\text{VOL}(A_t) + \text{VOL}(D_t)].$$

for all $t = 1, \ldots, |I|$ and all $I \in \mathcal{I}^{\text{on}}$. Here $\vec{S}$ is the sequence of solutions produced by A. Note that our definition of migration is amortized, i. e. we can build a potential to use later on. This will essentially allows us to repack the complete instance from time to time. In contrast, in the notion of worst-case migration, the total size of all repacked items is at most $\beta \cdot v_t$ at each time t, i. e. one cannot save up migration for later use.

If A is $(\gamma + \epsilon)$-competitive for some constant γ and all $\epsilon > 0$, and additionally has migration factor bounded by $f(1/\epsilon)$ for some function f, we say that A is *robust*. This is due to the fact that the migration needed only depends on the desired quality of the solution.

2.2.2 The General Framework

Before we start looking at the specific problems, we will introduce our very general framework. The simple algorithm presented by Feldkord *et al.* [47] can be seen as a special case of our framework. By using the concept of amortized migration, we can save up repacking potential, to be used at a later time. Now the design of an algorithm boils down to three basic questions. How do we pack arriving items? How do we repack arriving items? And at what time do we repack items? The main idea behind our framework for packing problems is to use general algorithms for the first two problems: an online algorithm to pack arriving items and an offline algorithm for the repacking. The third problem is then solved by a generic combination of the two algorithms. In order for this approach to work, we need different criteria for both algorithms and the packing problem that we are trying to solve. A simpler version of this framework was also used Feldkord *et al.* [47] for the Bin Packing problem.

Definition 2.2.2. Let Π be an minimization problem with sizes v_i. We call Π *space related*, if $\text{OPT}(I) \geq \text{VOL}(I)$ instances $I \in \Pi$. Here $\text{VOL}(I) := \sum_{i \in I} v_i$ is the total size of I.

Intuitively, this definition captures the fact that Π is a packing problem that needs to pack items of a certain volume in some non-overlapping way. All problems discussed here will be space related. To make use of this relation, we also need online algorithms such that competitiveness not only holds for the optimum $\text{OPT}(I)$, but also for the volume $\text{VOL}(I)$. We therefore also formally introduce this necessity.

Definition 2.2.3. Let Π be an online minimization problem with sizes v_i and A be an online algorithm for Π. We say that A is *space related with ratio* β, if

$$\text{COSTS}(S_t) \leq \beta \cdot \text{VOL}(I_t) + c$$

for all online instances $I \in \mathcal{I}^{\text{on}}$ and all time points $t = 1, 2, \ldots, |I|$. Here, S_t denotes the solution produced by A at time t.

Trivially, a space related algorithm with ratio β for a space related problem implies β-competitiveness, like Next-Fit for the Bin Packing problem. As indicated above, we will combine such an online algorithm with another offline algorithm. To be able to efficiently combine these two algorithms the online algorithms need to be able to build flexibly on top of the solution of the offline algorithm.

More formally, we require that our online algorithm takes another optional argument $S \in \text{SOL}(I_t)$ describing an existing solution to the previous instance I_t to build upon.

Definition 2.2.4. Let Π be an online minimization problem with sizes v_i and A be a space related online algorithm for Π with ratio β. Furthermore let $I \in \mathcal{I}^{\text{on}}$ be a static online instance (i. e. it contains no departures), $t < t' \leq |I|$ two time points, and S be a solution of I_t. We say that A is *flexible*, if it also accepts S as another parameter and produces upon input $I_{t'}$ and S a solution S' with $S'(i) = S(i)$ for all $i \in I_t \cap I_{t'}$.

Note that we define flexibility only with regard to static online instances where no items depart. One advantage of our framework is that we only need to design such online algorithms, but our combined algorithm will also be able to deal with departures. Remember that our online algorithm A is given some solution S to the previous items I_t. In order to be able to ignore the departure of items in the combined approach, we need to guarantee that A only introduces an error of $\beta[\text{VOL}(I_{t'}) - \text{VOL}(I_t)]$ when it packs instance $I_{t'}$.

Definition 2.2.5. We say that A is *flexible with ratio* β, if A is flexible and

$$\text{A}(I_{t'}, S) \leq \text{COSTS}(S) + \beta[\text{VOL}(I_{t'}) - \text{VOL}(I_t)] + c$$

for some constant c for all instances $I \in \mathcal{I}^{\text{on}}$, all $t \in \{1, \ldots, |I|\}$, and all $t' \in \{t + 1, \ldots, |I|\}$.

For the problems we will address later, Bin Packing and Strip Packing, we can simply solve the instance containing the newly arriving jobs separately and pack the new partial solution on top of the old one. Therefore this will be a property easily fulfilled by all online algorithms that we consider later. Note that a flexible algorithm with ratio β is also space related with ratio β, as we can simply choose $t = 1$ and a trivial solution S for this instance with a single item.

Combining the Algorithms

In order to obtain a robust PTAS or a robust online algorithm that is $\gamma + \epsilon$-competitive, we will need a flexible online algorithm with a constant ratio. For the offline algorithm we will need an offline $\gamma + \epsilon$-approximation (in the case of $\gamma = 1$, this is simply a PTAS). Basically, our final algorithm will have the same ratios as the offline algorithm and migration factor $O(1/\epsilon)$. In order to achieve a bounded competitiveness, we need to migrate items at some special time points. These time points will be determined by the total volume of items that arrived or departed since the last such

LISTING 2.1: Framework $\mathsf{ALG}(\mathsf{A}_{\text{on}}, \mathsf{A}_{\text{off}})$

```
let V_total := 0;
let V_changed := 0;
let S be an empty solution;
for each time t and arriving or departing item i
  if item i arrives do
    pack i according to online algorithm A_on and solution S;
  endif
  if item i departs do
    leave i in its position in the current solution S;
  endif
  V_changed = V_changed + v_i;
  if V_changed > εV_total do
    Compute offline solution S for I_t with A_off;
    Continue with new solution;
    Set V_total = VOL(I_t);
    Set V_changed = 0;
  endif
```

time point. At these special time points, we will use the offline algorithm to rebuild the solution completely. In between these points, we will only apply the online algorithm for the static case. We will therefore define phases such that during a phase we only apply the online algorithm.

Definition 2.2.6. Let Π be an online minimization packing problem with sizes v_i and $I \in \mathcal{I}^{\text{on}}$ be an online instance.

We partition $I_1, I_2, \ldots$ into *phases* as follows: The start time of the first phase is 1. If τ is the start time of the current phase, and $t \geq \tau$ is some time point, we define the following values: (i) the complete volume of the instance at time τ is denoted by $V_\tau = \text{VOL}(I_\tau)$, (ii) the items inserted since τ are defined as $\text{Ins}_t = \bigcup_{i=\tau}^{t-1}(I_{i+1} \setminus I_i)$ and its volume is $A_t = \text{VOL}(\text{Ins}_t)$, and (iii) the items departed since τ are defined as $\text{Dep}_t = \bigcup_{i=\tau}^{t-1}(I_i \setminus I_{i+1})$ and its total volume is $R_t = \text{VOL}(\text{Dep}_t)$. The current phase ends at the earliest point of time $\tau' > \tau$ such that $A_{\tau'} + R_{\tau'} > \epsilon V_\tau$. The next phase then starts at time τ'.

Basically we end a phase when the total size of items that were added or removed exceeds an ϵ factor of the total item size at the beginning of the phase. So our final algorithm basically packs inserted items using the online algorithm until a phase ends. Departed items stay at the same position. Whenever a phase ends, we compute an offline solution for the current instance and repack our solution completely. After that the next phase starts, where we again only pack with the online algorithm. Now we will prove that this combination of algorithms will be a robust $\gamma + \epsilon$ algorithm for $\gamma \in O(1)$ and fixed $\epsilon > 0$, when we combine two algorithms with the right properties.

Theorem 2.2.1. *Let Π be a minimization problem fulfilling Assumption 1. Furthermore let A_{off} be a $(\gamma + \epsilon)$-approximation algorithm for the offline version of Π for some constant $\gamma \in O(1)$, let $1/2 \geq \epsilon > 0$, and let A_{on} be a flexible algorithm with ratio β. Then the combination of these two algorithms, denoted with $\mathsf{ALG}(\mathsf{A}_{\text{on}}, \mathsf{A}_{\text{off}})$, is an $(\gamma + O(1)\beta\epsilon)$-competitive robust algorithm for Π with amortized migration factor $O(\frac{1}{\epsilon})$.*

The running time of ALG *at time point t is at most $T_{\text{off}}(t) + T_{\text{on}}(t)$, where $T_{\text{off}}(t)$ (resp. $T_{\text{on}}(t)$) is the worst-case running time of* A_{off} *(resp.* A_{on}*) on an instance of t items.*

Proof. Migration factor: Let I_τ be the instance at the start of a phase with volume $V_\tau = \text{VOL}(I_\tau)$ and let τ' be the ending time of this phase. As the phase ends at time τ', we have $A_{\tau'} + R_{\tau'} > \epsilon V_\tau$ (where $A_{\tau'}$ and $R_{\tau'}$ are defined as in Definition 2.2.6). As we only ever migrate items at the end of a phase, we only need to consider the amortized migration factor at these time points. We assign the volume of all items inserted and departed during the phase that ends at time τ' to this phase. Hence, a total migration potential of $A_{\tau'} + R_{\tau'}$ was build up in this phase. The total volume of the instance at this point is at most $V_\tau + A_{\tau'}$ and our amortized migration factor is thus

$$\frac{V_\tau + A_{\tau'}}{A_{\tau'} + R_{\tau'}} \leq \frac{V_\tau}{A_{\tau'} + R_{\tau'}} + 1 < \frac{V_\tau}{\epsilon V_\tau} + 1 = O(1/\epsilon).$$

As the phases are disjoint, the total amortized migration factor is thus also bounded by $O(1/\epsilon)$.

Competitiveness: To show the competitiveness of ALG, assume that

$$\mathsf{A}_{\text{off}}(I_t) \leq (\gamma + \epsilon) \cdot \text{OPT}(I_t) + c_{\text{off}} \tag{$*_{\text{off}}$}$$

for some constant c_{off} and furthermore

$$\mathsf{A}_{\text{on}}(I_{t'}, S) \leq \text{COSTS}(S) + \beta[\text{VOL}(I_{t'}) - \text{VOL}(I_t)] + c_{\text{on}} \tag{$*_{\text{on}}$}$$

for some constant c_{on}.

Let $I \in \mathcal{I}^{\text{on}}$ be some online instance and $t \in \{1, \ldots, |I|\}$. We distinguish whether t is the end of a phase or in the middle of a phase. If $t = \tau$ is the end of a phase, we use the offline algorithm A_{off} and thus have

$$\mathsf{ALG}(I_t) = \mathsf{A}_{\text{off}}(I_t) \leq (\gamma + \epsilon) \cdot \text{OPT}(I_t) + c_{\text{off}},$$

where the inequality follows from eq. ($*_{\text{off}}$). Now consider any point of time t during the phase starting at τ. Like above, let A_t, R_t denote the total volumes of arrived and removed items in this phase up time t. Note that this time $A_t + R_t \leq \epsilon V_\tau$, since otherwise we would repack.

Claim 1. *We have* $\text{OPT}(I_\tau) \leq \text{OPT}(I_t) + \beta\epsilon V_\tau + c_{\text{on}}$.

Proof. By Assumption 1, the value $\text{OPT}(I_t)$ is minimal if only departures happened. We can thus assume w. l. o. g. that up till time t some items were removed and no new items arrived. Consider an optimal solution S_t for the instance I_t, where items departed and let $D_t = I_\tau \setminus I_t$ be the set of departed items. Let $d_1, \ldots, d_k$ be some total ordering of D_t. We now construct a new online instance I' that somehow reverses the removal of D_t. The instance I' is of length $t + k$ with $I'_i = I_i$ for $i \leq t$. For $i > t$, we define $I'_i = I_{i-1} \cup \{d_i\}$. We now use the online algorithm A_{on} on this instance I' with solution S_t for instance I_t. At the end of instance I', eq. ($*_{\text{on}}$) implies that A_{on}

gives a feasible solution to $I'_{t+k} = I_t \cup D_t = I_\tau$ with

$$\begin{aligned}\mathsf{A}_{\text{on}}(I'_{t+k}, S_t) \leq\ & \text{COSTS}(S_t) + \beta[\text{VOL}(I_{t+k}) - \text{VOL}(I_t)] + c_{\text{on}} = \\ & \text{OPT}(I_t) + \beta[\text{VOL}(I_\tau) - \text{VOL}(I_t)] + c_{\text{on}} = \\ & \text{OPT}(I_t) + \beta\,\text{VOL}(D_t) + c_{\text{on}} \leq \\ & \text{OPT}(I_t) + \beta\epsilon V_\tau + c_{\text{on}}.\end{aligned}$$

The last inequality follows from the fact that $R_t = \text{VOL}(D_t)$ by definition and $R_t \leq \epsilon V_\tau$ by assumption. As A_{on} produces a feasible solution for $I'_{t+k} = I_\tau$, we clearly have $\text{OPT}(I_\tau) \leq \mathsf{A}_{\text{on}}(I'_{t+k}, S_t)$ thus proving the claim. □

Since the problem Π is space related, we can conclude that

$$\text{OPT}(I_t) \geq \text{VOL}(I_t) = V_\tau + A_t - R_t \geq V_\tau - R_t \geq V_\tau - \epsilon V_\tau \geq \frac{V_\tau}{2}. \qquad (*)$$

Hence $V_\tau \leq 2\,\text{OPT}(I_t)$.

Let S_τ be the solution produced by A_{off} at time τ that the online algorithm A_{on} is building upon. Note that we do only remove the departed item at a time point where the offline algorithm is used. Hence, at time t, the online algorithm does not produce a solution for the instance I_t, where the departed items are already removed. The algorithm rather works on the instance that still contains all items that have departed since time τ. This instance is denoted by I'_t. We thus have

$$\begin{aligned}&\mathsf{ALG}(I_t) = \mathsf{A}_{\text{on}}(I'_t, S_\tau) \leq && \text{// eq. } (*_{\text{on}})\\ &\text{COSTS}(S_\tau) + \beta[\text{VOL}(I'_t) - \text{VOL}(I_\tau)] + c_{\text{on}} = && \text{// } S_\tau \text{ produced by } \mathsf{A}_{\text{off}}\\ &\mathsf{A}_{\text{off}}(I_\tau) + \beta[\text{VOL}(I'_t) - \text{VOL}(I_\tau)] + c_{\text{on}} \leq && \text{// eq. } (*_{\text{off}})\\ &(\gamma+\epsilon)\cdot\text{OPT}(I_\tau) + c_{\text{off}} + \beta[\text{VOL}(I'_t) - \text{VOL}(I_\tau)] + c_{\text{on}} \leq && \text{// Claim 1}\\ &(\gamma+\epsilon)\cdot[\text{OPT}(I_t) + \beta\epsilon V_\tau + c_{\text{on}}] + c_{\text{off}} + \beta[\text{VOL}(I'_t) - \text{VOL}(I_\tau)] + c_{\text{on}} = &&\\ &(\gamma+\epsilon)\cdot[\text{OPT}(I_t) + \beta\epsilon V_\tau + c_{\text{on}}] + c_{\text{off}} + \beta A_t + c_{\text{on}} \leq && \text{// } A_t + R_t \leq \epsilon V_\tau\\ &(\gamma+\epsilon)\cdot[\text{OPT}(I_t) + \beta\epsilon V_\tau + c_{\text{on}}] + c_{\text{off}} + \beta\epsilon V_\tau + c_{\text{on}} = &&\\ &(\gamma+\epsilon)\,\text{OPT}(I_t) + (\gamma+\epsilon+1)\beta\epsilon V_\tau + (\gamma+\epsilon+1)c_{\text{on}} + c_{\text{off}} \leq && \text{// eq. } (*)\\ &(\gamma+\epsilon)\,\text{OPT}(I_t) + 2(\gamma+\epsilon+1)\beta\epsilon\,\text{OPT}(I_t) + (\gamma+\epsilon+1)c_{\text{on}} + c_{\text{off}} = &&\\ &(\gamma+\epsilon+2(\gamma+\epsilon+1)\beta\epsilon)\,\text{OPT}(I_t) + (\gamma+\epsilon+1)c_{\text{on}} + c_{\text{off}}. &&\end{aligned}$$

As γ, c_{on}, and c_{off} are constants, the last term can be written as

$$\begin{aligned}&(\gamma+\epsilon+2(\gamma+\epsilon+1)\beta\epsilon)\,\text{OPT}(I_t) + (\gamma+\epsilon+1)c_{\text{on}} + c_{\text{off}} \leq\\ &(\gamma + O(1)\cdot\beta\epsilon))\,\text{OPT}(I_t) + O(1).\end{aligned}$$

The running time bound follows easily from the fact that we essentially only use A_{on} or A_{off} at any given time. □

2.3 2-Dimensional Strip Packing

In the *online Strip Packing* problem, we are given a two-dimensional strip of width 1 and infinite height. At time t, either a rectangle r_t with width $w(r_t) \leq 1$ and height $h(r_t) \leq 1$ is inserted and needs to be packed into this strip or a rectangle r_t is removed from the strip. A packing is valid if no two rectangles intersect. The size $v(r)$ of a rectangle r is defined as $v(r) = h(r) \cdot w(r)$. We first focus on the case that

rectangles are not allowed to be rotated. A simple adaption of our algorithm also handles the case, where rotations by 90 degree are allowed. In both cases, the goal is to minimize the height of the produced packing. This problem has been studied intensively in the online setting (see for example the works cited in [32]). Jansen *et al.* [65] studied the static case in the migration scenario, where rectangles can only arrive.

To use our framework, we need the following ingredients:

i) We need to show that the Strip Packing problem is space related;

ii) we need to construct a flexible online algorithm with ratio β;

iii) We need to construct an offline approximation algorithm.

Concerning the first point, it is well-known that $\text{OPT}(I_t) \geq \text{VOL}(I_t)$, as the rectangles are not allowed to intersect and the width of the strip is exactly 1.

Remark 1. The Strip Packing problem is space related.

We will now present a flexible online algorithm with ratio $\beta = 4$. This algorithm is a simple adaption of the *shelf algorithms* presented by Baker and Schwarz [4]. In the notion of Csirik and Woeginger [35], this algorithm would be denoted as $\text{SHELF}(\mathsf{FirstFit}, 1/2)$. For the sake of completeness, we give a self-contained description and analysis. We first define several types of containers. A container c of type γ_0 has width 1 and height $h(c) = 1$. For $i \in \mathbb{N}_{\geq 1}$, a container of type γ_i has width 1 and height $h(c) = 2^{-i+1}$. For each $i \in \mathbb{Z}_{\geq 0}$, we will have at most one *active* container of type γ_i. For all other containers of this type – which we call *closed* – we will guarantee that at least $1/4$ of their volume is used by items.

We perform the following operation whenever a new rectangle r_t arrives:

- If $w(r_t) \geq 1/2$, check whether a container of type γ_0 exists. If not, open a new active container of type γ_0 and place r_t into it. If such a container c already exists and $h(r_t) + \sum_{r \in c} h(r) > 1$, declare c as closed and open a new active container of type γ_0. Otherwise ($h(r_t) + \sum_{r \in c} h(r) \leq 1$), put r_t on top of the top item in c.
- If $w(r_t) \leq 1/2$ and $h(r_t) \in (2^{-i}, 2^{-i+1}]$, check whether a container of type γ_i exist. If not, open a new active container of type γ_i and place r_t into it. If such a container c already exists and $w(r_t) + \sum_{r \in c} w(r) > 1$, declare c as closed and open a new active container of type γ_i. Otherwise ($w(r_t) + \sum_{r \in c} w(r) \leq 1$), put r_t right to the right-most item in c.

We have at most one active container of type γ_i for each $i \in \mathbb{Z}_{\geq 0}$: we only open a new active container if we simultaneously declare another container of the same type as closed.

We also have the following simple lemma showing that the volume of closed containers can be bounded.

Lemma 2.3.1. *Let t be any time point, $i \in \mathbb{Z}_{\geq 0}$, and $c_1, \ldots, c_k$ be the containers of type γ_i at time t. We have*

$$\sum_{j=1}^{k} \sum_{r \in c_j} v(r) \geq 1/4 \big(\sum_{j=1}^{k} h(c_j) \big) - 2^{-i+1}.$$

Proof. If $k = 1$, only one container of type γ_i exists. As $h(c_j) \leq 2^{-i+1}$, the inequality holds. The left hand side is strictly larger than 0 and the right hand side is strictly smaller than 0.

If $k > 1$, we can argue about the contained volume. If $i = 0$, we have opened container $j > 1$, as $\left[\sum_{r \in c_{j-1}} h(r)\right] + \left[\sum_{r \in c_j} h(r)\right] > 1$. As all of these rectangles were placed in a container of type γ_0, they have width at least $1/2$. Summing up, as $h(c_j) = 1$, this gives

$$\sum_{j=1}^{k} \sum_{r \in c_j} v(r) \geq \sum_{j=0}^{\lfloor k/2 \rfloor - 1} \Big[\sum_{r \in c_{2j+1}} v(r) \Big] + \Big[\sum_{r \in c_{2j+2}} v(r) \Big] =$$

$$\sum_{j=0}^{\lfloor k/2 \rfloor - 1} \Big[\sum_{r \in c_{2j+1}} h(r) \cdot \underbrace{w(r)}_{\geq 1/2} \Big] + \Big[\sum_{r \in c_{2j+2}} h(r) \cdot \underbrace{w(r)}_{\geq 1/2} \Big] \geq$$

$$(1/2) \sum_{j=0}^{\lfloor k/2 \rfloor - 1} \underbrace{\Big[\sum_{r \in c_{2j+1}} h(r) \Big] \Big[\sum_{r \in c_{2j+2}} h(r) \Big]}_{>1} > (1/2)[\lfloor k/2 \rfloor] \geq (1/2)(k/2 - 1) =$$

$$(1/4)k - (1/2) = (1/4) \sum_{j=1}^{k} h(c_j) - (1/2) > (1/4) \sum_{j=1}^{k} h(c_j) - \underbrace{2}_{=2^{-0+1}}.$$

If $i > 0$, we have opened container c_j for $j > 1$, as $\left[\sum_{r \in c_{j-1}} w(r)\right] + \left[\sum_{r \in c_j} w(r)\right] > 1$. As all of these rectangles were placed in a container of type γ_i, they have height at least 2^{-i}. Summing up, as $h(c_j) = 2^{-i+1}$, this gives

$$\sum_{j=1}^{k} \sum_{r \in c_j} v(r) \geq \sum_{j=0}^{\lfloor k/2 \rfloor - 1} \Big[\sum_{r \in c_{2j+1}} v(r) \Big] \Big[\sum_{r \in c_{2j+2}} v(r) \Big] =$$

$$\sum_{j=0}^{\lfloor k/2 \rfloor - 1} \Big[\sum_{r \in c_{2j+1}} \underbrace{h(r)}_{>2^{-i}} \cdot w(r) \Big] + \Big[\sum_{r \in c_{2j+2}} \underbrace{h(r)}_{>2^{-i}} \cdot w(r) \Big] \geq$$

$$2^{-i} \sum_{j=0}^{\lfloor k/2 \rfloor - 1} \underbrace{\Big[\sum_{r \in c_{2j+1}} w(r) \Big] + \Big[\sum_{r \in c_{2j+2}} w(r) \Big]}_{>1} >$$

$$2^{-i} \lfloor k/2 \rfloor \geq 2^{-i}(k/2 - 1) = (1/4)2^{-i+1} \cdot k - 2^{-i} =$$

$$(1/4) \sum_{j=1}^{k} h(c_j) - 2^{-i} \geq (1/4) \sum_{j=1}^{k} h(c_j) - 2^{-i+1}. \qquad \square$$

If we are given a previous packing S, we simply build the containers on top of it. Clearly, this gives a solution of height $h(S) + \sum_{j=1}^{k} h(c_j)$, if $c_1, \ldots, c_k$ are the containers constructed by the algorithm. We will now use Theorem 2.3.1 to show that this algorithm is indeed a flexible online algorithm with ratio 4.

Theorem 2.3.2. *The presented algorithm A_{SP} is a flexible online algorithm for Strip Packing with ratio 4.*

Proof. The flexibility of A_{SP} follows directly due to the fact that we only build upon the existing packing. We will now show that A_{SP} has ratio 4. Let t be any time point, S be the previous packing we built upon, and $c_1, \ldots, c_k$ be the containers constructed at time t. The set of rectangles contained in the containers is denoted as J. As note above, the current packing has height $h(S) + \sum_{j=1}^{k} h(c_j)$. For $i \in \mathbb{Z}_{\geq 0}$, let $C_i \subseteq \{c_1, \ldots, c_k\}$ be the set of containers of type γ_i. We then have $\sum_{j=1}^{k} h(c_j) =$

$\sum_{i\geq 0}\sum_{c\in C_i} h(c)$. Using Theorem 2.3.1 gives

$$\sum_{i\geq 0}\sum_{c\in C_i} h(c) \leq \sum_{i\geq 0}\big[\sum_{c\in C_i} h(c)\big] - 2^{-i+3} + 2^{-i+3} = \big(\sum_{i\geq 0}\big[\sum_{c\in C_i} h(c)\big] - 2^{-i+3}\big) + \sum_{i\geq 0} 2^{-i+3} \leq$$
$$\big(\sum_{i\geq 0}\big[\sum_{c\in C_i} h(c)\big] - 2^{-i+3}\big) + 16 = 4\big(\sum_{i\geq 0}\big[\sum_{c\in C_i} (1/4)h(c)\big] - 2^{-i+1}\big) + 16 \underbrace{\leq}_{\text{Theorem 2.3.1}}$$
$$4\big(\sum_{i\geq 0}\sum_{c\in C_i}\sum_{r\in C} v(r)\big) + 16 = 4\,\text{VOL}(J) + 16. \qquad \square$$

We have now shown the first two ingredients for our framework: the problem is space related and we gave a suitable online algorithm. The final piece – an offline approximation algorithm – is given by the asymptotic fully polynomial time approximation scheme (AFPTAS) of Kenyon and Rémila [70], which is an $1+\epsilon$-approximation. We can thus use Theorem 2.2.1 with $\gamma = 1$ and $\beta = 4$ to conclude the following theorem.

Theorem 2.3.3. *There is a robust online algorithm for the dynamic Strip Packing problem that is $1+\epsilon$-competitive and has amortized migration factor $O(1/\epsilon)$.*

Rotations

If rotations by 90 degree are allowed, the resulting problem is called Strip Packing With Rotations. For an instance I, we denote the height of a corresponding optimal packing by $\text{OPT}_R(I)$. As the volume of a rotated rectangle does not change, we have $\text{OPT}_R(I) \geq \text{VOL}(I)$. Similarly, the volume bound of Theorem 2.3.1 also remains true. We can thus conclude the following adaption of Theorem 2.3.2.

Theorem 2.3.4. *The presented algorithm A_{SP} is a flexible online algorithm for Strip Packing With Rotations with ratio 4.*

Instead of using the classical AFPTAS by Kenyon and Rémila [70], we use the AFPTAS of Jansen and van Stee [64] for the case that rotations are allowed. Using Theorem 2.2.1 with $\gamma = 1$ and $\beta = 4$ gives the following theorem.

Theorem 2.3.5. *There is a robust online algorithm for the dynamic Strip Packing With Rotations problem that is $1+\epsilon$-competitive and has amortized migration factor $O(1/\epsilon)$.*

The best known online algorithm with migration known is due to Jansen *et al.* [65]. It also is $1+\epsilon$-competitive, but an amortized migration factor of $O(1/\epsilon^9 \log^2(1/\epsilon))$, only works for the static case (no rectangles are removed), and cannot handle rotations.

2.4 Bin Packing

2.4.1 2-Dimensional Bin Packing

In 2-D Bin Packing, each item i is given by its height $h_i \leq 1$ and its width $w_i \leq 1$. The goal is to pack these items non-overlapping into as few unit-sized squares (called *bins*) as possible. As above, we will show the following:

i) We need to show that the 2-D Bin Packing problem is space related;

ii) We need to construct a flexible online algorithm with ratio β;

iii) We need to construct an offline approximation algorithm.

As the rectangles are not allowed to overlap and each bin has a total volume of 1, 2-D Bin Packing is space related.

Remark 2. The 2-D Bin Packing problem is space related.

We will now present a flexible online algorithm. This algorithm is a simple extension of the classical algorithm presented by Coppersmith and Raghavan [34]. For the sake of completeness, we give a self-contained description and analysis. We categorize items as follows: We call an item *vertical* if $w_i \leq h_i$ and *horizontal* if $w_i > h_i$. Note that squares with $h_i = w_i$ will be considered as vertical. Without loss of generality we will explain in the following how to pack vertical items. Horizontal items can and will be placed into separate bins with the same strategy, just altered for horizontal items. Note that we later also need to account for these horizontal bins.

We further assign each item i a *size class*. Item i is in size class $j \in \mathbb{N}_{\geq 1}$ if $1/2^{j-1} \geq h_i > 1/2^j$. Further we say an item i is *square-like*, if i is in size class j and furthermore $w_i > 1/2^j$. The general idea is that for every arriving item in size class j, we will assign a square slot of size $1/2^{j-1}$ in some bin. A square slot of this size is called a *slot of class j*. Note that an item of size class j always fits into a slot of class j, as we only handle vertical items with $w_i \leq h_i$ here. Our goal is to fill all opened slots with items until $1/4$ of the total area of the slot is covered. Square-like items will immediately fill such a slot to this extent. For the other items, we will *reserve* such slots for a size class j and stack items from left to right until the total width of all items in that slot exceeds $1/2^j$. Since the height of items assigned to this slot exceed $1/2^j$ as well, $1/4$ of the total slot will be covered at that point. A slot can have three states: it is either (i) *empty* and thus contains no item, (ii) *reserved* for class j and thus only contains items of class j or (iii) *closed* if at least $1/4$ of its total volume is filled with items.

In order to assign items to these slots, we will keep up to two open bins. The first open bin will hold items of size class 1 and we use the complete bin as a single reserved slot. The second open bin will receive items of size class ≥ 2. Initially, this bin is split into four empty slots of class 2. We now define our online algorithm $A_{2\text{-}D}$:

1. If a square-like item of size class 1 arrives, we open a new bin, place the item in there and close it immediately. By definition of size class 1, at least $1/4$ of the volume of this bin will be filled.

2. If a non-square-like item i of size class 1 arrives, we place it to the right of the non-square-like items in the first open bin or on the left end, if no such items exist. As $w_i \leq 1/2$, we can always place this item in the bin. If the sum of the widths of the items in the first bin now is at least $1/2$, close this bin and open a new first bin. As every item placed in the bin has height $h_i \geq 1/2$, at least $1/4$ of the volume of this bin will be filled.

3. If an item i of class $j \geq 2$ arrives, find a non-closed slot of class $j' \leq j$ with maximal j' in the second open bin.

   (a) If $j' = j$, we choose a slot reserved for j' or – if no such slot exists – the top-most left-most of the slots of class j'. We declare this slot as reserved for j' and place the item right of the items placed in this slot or on the left end, if no such items exist. As $w_i \leq h_i \leq 1/2^{j-1}$, we can always place i in this slot. If the sum of the widths of the items placed in this slot is at least $1/2^j$, we close this slot.

(b) If $j' < j$, note that we can split such a slot into four equal-sized slots of class $j' + 1$. Splitting one slot will leave us with four slots: three of the slots will remain empty and the last slot will either be split even more (if its class $j' + 1$ is smaller than j) or we reserve this slot for class j and put the new item into it. We repeat this splitting until we have created a slot of class j and put i on the left end. If the sum of the widths of the items placed in this slot is at least $1/2^j$ (i. e. if i is square-like), we close this slot.

(c) If no slot of size $j' \leq j$ exists, we close the second bin and open a new second bin containing four empty slots of class 2. We place i into this bin as above.

We will now show that every bin created by the algorithm contains a most three empty slots of a certain class.

Lemma 2.4.1. *For each $j \geq 1$, every bin created by $\mathsf{A}_{2\text{-}D}$ contains at most three empty slots of class j.*

Proof. For $j = 1$, every bin contains at most a single slot of class 1.

For $j \geq 2$, assume that this is not the case and our algorithm may generate a slot assignment with four or more slots of the same class j^*. Consider some arriving item i of class j such that before the arrival of i there are at most three empty slots of class j^* and afterwards, there are four or more of these slots. Since the number of empty slots of class j^* increased, our algorithm must have split a slot of class $j' < j^*$. Note that this will only happen if $j^* \leq j$. Splitting this slot of class j' over and over will create three slots of class $j' + 1, j' + 2, \ldots, j^* - 1$. Finally, four slots of class j^* are created. One of this slots will either be directly used for i (if $j^* = j$) or will be split into smaller slots. Hence, only three new slots of class j^* are created. By assumption, the insertion of i leads to at least four slots of class j^*. Hence, there must have been a slot of class j^* before. But, by definition of the algorithm, we would rather have chosen this slot instead of the slot of class j', as $j' < j^*$. Hence, this is a contradiction. □

To make the algorithm $\mathsf{A}_{2\text{-}D}$ flexible, we simply ignore all previous used bins and only work on the newly generated bins.

Theorem 2.4.2. *The proposed algorithm $\mathsf{A}_{2\text{-}D}$ for 2-D Bin Packing is a flexible online algorithm with ratio $\frac{48}{5}$.*

Proof. The flexibility of $\mathsf{A}_{2\text{-}D}$ follows directly from the fact that we only work on newly created bins.

To analyze the ratio of the algorithm we look at how much total area is covered when a complete bin has been closed. First we can observe that every closed slot inside a closed bin has at least an overall area of $\frac{1}{4}$ covered: we only close a slot if half of its width is covered and we only put items into a slot that cover at least half of the height.

Another easy observation is the fact that the algorithm may have at most one reserved slot for each class, since the algorithm only opens a new reserved slot for a class whenever the current reserved one is closed. We may open a second slot for a square-like item, but will also close it immediately. By neglecting the items in reserved slots, the unused space due to reserved slots in every bin therefore can be bounded by

$$\sum_{j=1}^{\infty} 1/4^j = \frac{1}{1 - 1/4} - 1 = \frac{1}{3}. \qquad (*)$$

By Theorem 2.4.1, we know that every bin contains at most three empty slots of each size class. Note that a closed bin cannot contain an empty slot of class 2, as such a bin will only receive items of this size class or higher. With this we can now conclude that the unused space due to empty slots in a closed bin is at most

$$3\sum_{i=2}^{\infty} 1/4^i = 3[\frac{1}{1-1/4} - 1 - 1/4] = 3 \cdot 1/12 = 1/4. \qquad (**)$$

By combining eq. (*) and eq. (**), at least a total area of $1 - 1/4 - 1/3 = 5/12$ of every closed bin is occupied by closed slots. As discussed earlier, every such closed slot has at least $1/4$ of its total area covered so we finally can conclude that at least a total area of $5/48$ of every closed bin must be covered.

The algorithm keeps at most four open bins, one for items of size class 1, one for items of other size classes, and we need two respective bins for horizontal items containing the same classes as well. Finally we get that our algorithm for an instance I with total volume $\text{VOL}(I)$ will use at most $48/5\,\text{VOL}(I) + 4$ bins in total. □

Finally, we can use the approximation algorithm of Bansal and Khan [7] that is an 1.405-approximation for 2-D Bin Packing. We can thus use Theorem 2.2.1 with $\gamma = 1.405$ and $\beta = 48/5$ to conclude the following theorem.

Theorem 2.4.3. *There is a robust online algorithm for the dynamic 2-D Bin Packing problem that is* $1.405 + \epsilon$*-competitive and has amortized migration factor* $O(1/\epsilon)$.

To the best of our knowledge, this is the first robust online algorithm for dynamic 2-D Bin Packing. Note that the best known lower bound for the competitiveness of any online algorithm for online 2-D Bin Packing without migration is 1.856 due to Van Vliet [81].

Rotations

As for Strip Packing, allowing rotations of the rectangles by 90 degrees gives rise to a problem called 2-D Bin Packing With Rotations. The corresponding optimal number of bins needed to pack instance I is denoted by $\text{OPT}_R(I)$. Rotations are invariant with regard to the volume of a rectangle and thus $\text{OPT}_R(I) \geq \text{VOL}(I)$. We can thus again use our online algorithm $\mathsf{A}_{2\text{-}D}$ to obtain the following adaption of Theorem 2.4.2.

Theorem 2.4.4. *The proposed algorithm* $\mathsf{A}_{2\text{-}D}$ *for 2-D Bin Packing With Rotations is a flexible online algorithm with ratio* $\frac{48}{5}$.

The approximation algorithm of Bansal and Khan [7] used above can also handle the case of rotation and thus is an 1.405-approximation for 2-D Bin Packing With Rotations. We can thus use Theorem 2.2.1 with $\gamma = 1.405$ and $\beta = 48/5$ to conclude the following theorem.

Theorem 2.4.5. *There is a robust online algorithm for the dynamic 2-D Bin Packing With Rotations problem that is* $1.405 + \epsilon$*-competitive and has amortized migration factor* $O(1/\epsilon)$.

To the best of our knowledge, this is the first robust online algorithm for dynamic 2-D Bin Packing With Rotations. Note that the best known lower bound for the competitiveness of any online algorithm for 2-D Bin Packing With Rotations without migration is at least 1.7515 due to Balogh *et al.* [5] improving the bound of 1.6707 due to Blitz *et al.* [18].

2.4.2 d-Dimensional Bin Packing

We will now look at the problem of packing d-dimensional hyperrectangles into as few unit-sized hypercubes as possible for higher dimensions $d > 2$. This problem is called *d-Dimensional Hyperrectangle Packing*.

To obtain an suitable flexible online algorithm for this problem, we will generalize the 2-dimensional Bin Packing algorithm $\mathsf{A}_{2\text{-}D}$ from Section 2.4.1 The side length of dimension $i \in \{1, \dots, d\}$ of a d-dimensional hyperrectangle r is denoted as r_i and its volume is $v(r) = \prod_{i=1}^{d} r_i$. Items will be classified almost the same as before in a straight-forward generalization. For a permutation $\pi\colon \{1, \dots, d\} \to \{1, \dots, d\}$, we associate a set of rectangles r with $r_{\pi(1)} \leq r_{\pi(2)} \leq \dots r_{\pi(d)}$ with it. We treat each of these permutations separately and pack the corresponding rectangles in separate bins. This will only give an additive error of $d!$ in our approximation guarantee. In the following, we thus fix a permutation π. A rectangle associated with π is in size class $j \in \mathbb{N}_{\geq 1}$, if $1/2^{j-1} \geq r_{\pi(d)} > 1/2^j$, hence the class of a rectangle depends on its largest side length. Rectangles of class j are packed into slots that are hypercubes with side length $1/2^{j-1}$. Such a slot is called a *slot of class j*. Note that an item of size class j always fits into a slot of class j, as we fixed the permutation π. We now proceed as in Section 2.4.1. If an item of class j arrives, we put it into an open slot of this class. If no such slot exists, we split a slot of class $j' \leq j$ into 2^d slots of class $j' + 1$ recursively until an empty slot of class j is created. We denote this straight-forward generalization of $\mathsf{A}_{2\text{-}D}$ as $\mathsf{A}_{d\text{-}D}$. It is easy to see that the d-dimensional analogue of Lemma 2.4.1 also holds.

Lemma 2.4.6. *For each π and each $j \geq 1$, every bin created by $\mathsf{A}_{d\text{-}D}$ contains at most $2^d - 1$ empty slots of class j associated with π.*

Every closed slot inside a closed bin has at least an overall area of 2^{-d} covered, as in the proof of Theorem 2.4.2. We thus obtain the following generalization of Theorem 2.4.2 by not using any bins created by the previous solution S.

Theorem 2.4.7. *The proposed algorithm $\mathsf{A}_{d\text{-}D}$ for d-Dimensional Hyperrectangle Packing is a flexible online algorithm with ratio $\frac{2^{2d}-3\cdot 2^d+1}{2^{2d}(2^d-1)}$.*

Proof Sketch. By neglecting the items in reserved slots, the unused space due to reserved slots in every bin can be bounded by

$$\sum_{j=1}^{\infty} 1/(2^d)^j = \frac{1}{1 - 1/2^d} - 1 = \frac{1}{2^d - 1}. \tag{*}$$

By Theorem 2.4.6, we know that every bin contains at most $2^d - 1$ empty slots of each size class for each π. With this we can now conclude that the unused space due to empty slots in a closed bin is at most

$$(2^d - 1) \sum_{i=2}^{\infty} 1/(2^d)^i = (2^d - 1)\Big[\frac{1}{1 - 1/2^d} - 1 - 1/2^d\Big] = (2^d - 1) \cdot 1/(2^d - 1) \cdot 2^d = 1/2^d. \tag{**}$$

By combining eq. (*) and eq. (**), at least a total area of $1 - 1/2^d - 1/2^{d-1} = \frac{2^{2d}-3\cdot 2^d+1}{2^d(2^d-1)}$ of every closed bin is occupied by closed slots. As discussed earlier, every such closed slot has at least $1/2^d$ of its total area covered so we finally can conclude that at least a total area of $\frac{2^{2d}-3\cdot 2^d+1}{2^{2d}(2^d-1)}$ of every closed bin must be covered. □

The best known offline algorithm for the d-dimensional Hyperrectangle Packing problem has ratio 1.69103^{d-1} and is due to Caprara [24] (see also [32]). We can thus use Theorem 2.2.1 with $\gamma = 1.69103^{d-1}$ and $\beta = \frac{2^{2d}-3\cdot 2^d+1}{2^{2d}(2^d-1)}$ to conclude the following theorem.

Theorem 2.4.8. *There is a robust online algorithm for the dynamic d-dimensional Hyperrectangle Packing problem that is $1.69103^{d-1} + \epsilon$-competitive and has amortized migration factor $O(1/\epsilon)$.*

Hypercube Packing

If every hyperrectangle is in fact a hypercube, we obtain a substantially easier problem, called *d-Dimensional Hypercube Packing*. The resulting algorithm is also much easier to describe. Here, a slot of class j is a hypercube of side length $1/2^{j-1}$. Putting an item of class j into a slot of class j covers at least a fraction of $1/2^d$ of the total volume of the slot. Hence, slots are either empty or closed and do not need to be reserved. Every d-dimensional hypercube with side length s can be divided into 2^d equal sized hypercubes with side length $s/2$. The slot assignment happens like above: We start with single empty active bin containing 2^d empty slots of class 2. An item of class 1 is simply put into its own bin. Whenever an item i of class $j \geq 2$ arrives, we first try to find an empty slot of class j. If any such slot exists, put i into it and close the slot. If no slot of class $j' \leq j$ exists, we close the bin and open a new bin to insert i into. If a slot of class $j' < j$ exists, we split the smallest such slot that is still larger than the required slot size until an empty slot of size j is created.

We call this algorithm $\mathsf{A}_{d\text{-hyper}}$ and again make it flexible by simply opening new bins. Similar to the two-dimensional case, we can show that there can't be too many empty slots of the same class. The proof is just an adaption of the proof of Theorem 2.4.1.

Lemma 2.4.9. *For each $j \geq 1$, every bin created by $\mathsf{A}_{d\text{-hyper}}$ contains at most $2^d - 1$ empty slots of class j.*

Proof. For $j = 1$, every bin contains at most a single slot of class 1.

For $j \geq 2$, assume that this is not the case and our algorithm may generate a slot assignment with 2^d or more slots of the same class j^*. Consider some arriving item i of class j such that before the arrival of i there are at most $2^d - 1$ empty slots of class j^* and afterwards, there are 2^d or more of these slots. Since the number of empty slots of class j^* increased, our algorithm must have split a slot of class $j' < j^*$. Note that this will only happen if $j^* \leq j$. Splitting this slot of class j' over and over will create $2^d - 1$ slots of class $j'+1, j'+2, \ldots, j^*-1$. Finally, 2^d slots of class j^* are created. One of this slots will either be directly used for i (if $j^* = j$) or will be split into smaller slots. Hence, only $2^d - 1$ new slots of class j^* are created. By assumption, the insertion of i leads to at least 2^d slots of class j^*. Hence, there must have been a slot of class j^* before. But, by definition of the algorithm, we would rather have chosen this slot instead of the slot of class j', as $j' < j^*$. Hence, this is a contradiction. □

Theorem 2.4.10. *The proposed algorithm $\mathsf{A}_{d\text{-hyper}}$ for d-Dimensional Hypercube Packing is a flexible online algorithm with ratio $\frac{2^{2d}}{2^d-1}$.*

Proof. The flexibility follows from the fact that we do not touch the already used bins.

Just like in the two dimensional case we analyze the free space in a closed bin. As observed above, every closed slot has at least an overall area of $1/2^d$ of its size covered. We now take a look at the empty slots in a closed bin.

In bins that were closed due to the insertion of an item of size class 1, at least a fraction of $1/2^d$ of the bins area is covered. Hence, consider a bin that received items with side length less or equal than $1/2$. As observed above, every assigned slot in this bin is either closed or empty. Theorem 2.4.9 guarantees that we have at most $2^d - 1$ empty slots of any given class. Since there can not be any reserved slots, we can already calculate how much space we lose in total, by looking at empty slots. Note again that all slots of class 2 will be used by some items before the overall bin gets closed. Therefore the total volume assigned to empty slots can be at most

$$(2^d - 1)\sum_{i=2}^{\infty} \frac{1}{\left(2^d\right)^i} = (2^d - 1)[\frac{1}{1 - 2^{-d}} - 1 - \frac{1}{2^d}] = (2^d - 1)[\frac{2^d}{2^d - 1} - 1 - \frac{1}{2^d}] =$$
$$2^d - 2^d + 1 - 1 + \frac{1}{2^d} = \frac{1}{2^d}.$$

We get that at least $1 - 1/2^d = 2^d - 1/2^d$ of the total area of a closed bins is used by closed slots. As each of these slots covers at least an area of $1/2^d$ of its size, at least a total volume of $2^d - 1/2^d \cdot 1/2^d = 2^d - 1/2^{2d}$ of a closed bin is covered. We also have to account that our algorithm may keep one open bin, to place items. Together we can conclude that our algorithm uses at most $2^{2d}/2^d - 1 \operatorname{VOL}(I) + 1$ bins in total. □

Finally, we can use the offline APTAS from Bansal *et al.* [8]. Using Theorem 2.2.1 with $\gamma = 1$ and $\beta = 2^{2d}/2^d - 1$ allows the conclusion of the following theorem.

Theorem 2.4.11. *For every $d \geq 2$, there is a robust online algorithm for the dynamic d-Dimensional Hypercube Packing problem that is $1 + \epsilon$-competitive and has amortized migration factor $O(1/\epsilon)$.*

In contrast, the best known online algorithm with migration for the d-Dimensional Hypercube Packing is due to Epstein and Levin [44]. It is also $1 + \epsilon$-competitive, but can only handle the static case and has migration factor $(1/\epsilon)^{\Omega(d)}$. Note however that they use worst-case migration, i. e. they are not allowed to repack the complete instance every once in a while but need to make slight adaptions carefully throughout the run of the algorithm.

2.5 d-Dimensional Strip Packing

In the d-dimensional version of online strip packing, called the *d-Dimensional Strip Packing* problem, we are given a d-dimensional cuboid that has size 1 in $d - 1$ dimensions and infinite size in the last dimension. Even in the general case with more dimensions we will consider this last dimension as *height*. At each point of time a d-dimensional hyperrectangle r arrives. The size of its ith dimension is denoted by $r_i \in (0, 1)$ and its volume is $v(r) = \prod_{i=1}^{d} r_i$. The task is again to pack these cuboid with no intersection such that the height is minimized. Like above we denote with I_t the set of rectangles present at time t,with $\operatorname{VOL}(I_t)$ the total volume and with $\operatorname{OPT}(I_t)$ the optimal height. This version of Strip Packing is also space related, since the base of the packing space has side lengths of 1.

Remark 3. The d-dimensional Strip Packing problem is space related.

It leaves to show that there are respective online and offline algorithms for our framework. For a flexible online algorithm we will generalize our above approach for the d-dimensional case in a straight-forward way. A *container* c of type γ_i is a d-dimensional hyperrectangle with $c_1 = c_2 = \ldots = c_{d-1} = 1$ and $c_d = 2^{-i}$. For each $i \in \mathbb{Z}_{\geq 0}$, we will have at most one *active* container of type γ_i. For all other containers of this type – which we call *closed* – we will guarantee that at least a constant fraction of their volume is used by items. We assign a hyperrectangle r of height $r_d \in (2^{-i-1}, 2^{-i}]$ to a container of type γ_i. We then treat these hyperrectangles as $d-1$-dimensional hyperrectangles by projecting to its first $d-1$ coordinates and also treat the container as a $d-1$-dimensional hypercube. We then pack the projected hyperrectangles into the projected hypercube with the algorithm $\mathsf{A}_{(d-1)\text{-}D}$ described in Section 2.4.2. Theorem 2.4.7 then guarantees that a fraction of $\frac{2^{2(d-1)}(2^{d-1}-1)}{2^{2(d-1)}-3\cdot 2^{d-1}+1}$ of each projected container is filled. As the original, non-projected containers have height 2^{-i} and every non-projected hyperrectangle has height at least 2^{-i-1}, we lose a factor of 2 here. The resulting algorithm, called A_{d-SP}, is thus a flexible online algorithm and we obtain the following generalization of Theorem 2.3.2.

Theorem 2.5.1. *The presented algorithm A_{d-SP} is a flexible online algorithm for d-dimensional Strip Packing with ratio $O(\frac{2^{2d}-3\cdot 2^d+1}{2^{2d}(2^d-1)})$.*

To the best of our knowledge, there is no work that explicitly deals with the construction of approximation algorithms for the d-dimensional Strip Packing problem. As shown above, any such result can be used in the context of our framework to obtain a robust online algorithm with corresponding competitive ratio.

2.5.1 Hypercube Strip Packing

In the following we will restrict this problem to the case where each item is a hypercube, so we have that each entry r_i is the same. In this version, that is also known as *online Hypercube Strip Packing*, we will denote this value with $s(r)$ and call it the *side length* of the hypercube r. The size of this hypercube is given by its volume, which is $v(r) = s(r)^d$. Note that in this variant allowing rotations by 90 degree makes no difference.

In fact the approach becomes a little easier due to our restriction to hypercubes. Again we define containers: For $i \in \mathbb{N}_{\geq 1}$, a container c of type γ_i has size 1 in every dimension except the height and height of $h(c) = 2^{-i+1}$. The idea is the same as before: Keep at most one *open* container of each size, while we make sure at least 2^{-d} of the total area of each *closed* bin is covered.

The assignment of items to containers works the same as well: Whenever a hypercube c_t with side length $s(c_t) \in (2^{-i}, 2^{-i+1}]$ arrives , we will try to pack it into an open container of type γ_i. If there is no open container of this type or it does not fit, we open a new one and close the old one. In order to place these hypercubes inside of the container we subdivide every container into equal sized cubic slots. Given a container of type γ_i, we further divide it into hypercubes with side length 2^{-i+1}, which we call *slots*. Now when a hypercube c_t with $s(c_t) > 2^{-i}$ gets assigned to a container, we pack it into one of the free slots, such that it does not intersect any other slots.

Overall this yields a space related online algorithm. The flexibility follows like in the 2-D case from the fact, that we can simply extend any existing packing.

Theorem 2.5.2. *The presented algorithm $\mathsf{A}_{SP\text{-}hyper}$ is a flexible online algorithm for d-dimensional Hypercube Strip Packing restricted to hypercubes with ratio 2^d.*

Proof. The flexibility follows again from the properties of the Strip Packing problem, given a packing of some items S, we can simply ignore that packing and put newly arriving items on top of the existing packing.

First off we show that each closed container has at least 2^{-d} of its total volume covered. Let c be a closed container of type γ_i with volume $v(c) = h(c)$, since the size in every dimension except height is exactly 1. Note that we subdivided c into multiple cubic slots with total volume $h(c)^d = 2^{d(-i+1)}$. Since c is closed each slot got assigned an item r with volume $v(r) = s(r)^d > 2^{-i \cdot d} = 2^{-d} \cdot 2^{-d \cdot i + d} = 2^{-d} \cdot h(c)^d$. Since the slots divide the container completely and every slot has 2^{-d} of its total volume covered, we can conclude that $\sum_{r \in c} v(r) \geq (2^{-d}) h(c)$.

Now we will look at the height of a flexibly created solution. Let t be a point of time, S be the previous packing we extended and let $c_1, .., c_k$ be the containers created up to time t. Let O be the set of open containers and L the set of closed containers. Let S_t be the result of our algorithm at time t. Let l be the type of the smallest open container in our current solution. We then can observe for the height of open containers that: $\sum_{c \in L} h(c) \leq \sum_{i=1}^{l} 2^{-i+1} = \sum_{i=0}^{l} 2^{-i} \leq 2$.

Finally we have for the total height of our solution that

$$h(S_t) = h(S) + \sum_{c \in L} h(c) + \sum_{c \in O} h(c) \leq h(S) + \sum_{c \in L} 2^d \sum_{r \in c} v(r) + \sum_{c \in O} h(c) \leq$$
$$h(S) + 2^d \sum_{c \in L} \sum_{r \in c} v(r) + 2 \leq h(S) + 2^d \operatorname{VOL}(I_t) + 2 \qquad \square$$

With this we know our problem is space-related and has an appropriate space related online algorithm. As for the offline algorithm, we will use a result from Harren, who gave an APTAS for the Hypercube Strip Packing problem [56]. By using Theorem 2.2.1 with $\gamma = 1$ and $\beta = 2^d$ we get the following theorem.

Theorem 2.5.3. *There is a robust online algorithm for the dynamic d-dimensional Hypercube Strip Packing problem that is* $1 + \epsilon$*-competitive and has amortized migration factor* $O(1/\epsilon)$.

2.6 Vector Packing

In the *online d-dimensional Vector Packing* problem, at time t either a vector $w_t \in (\mathbb{Q} \cap [0,1])^d$ is inserted and needs to be packed or is removed. The size $v(w_t)$ of such a vector $w_t = (w[1], \ldots, w[d])$ is defined as the average sum of its components, i. e. $v(w_t) = \sum_{j=1}^{d} w[j]/d$. The goal is to pack these vectors into as few as possible bins as possible. Here, a *bin* B is a subset of vector such that $\sum_{w \in B} w[j] \leq 1$ for $j = 1, \ldots, d$.

To use our framework, we need the following ingredients:

i) We need to show that the problem is space related;

ii) we need to construct a flexible online algorithm with ratio β;

iii) We need to construct an offline approximation algorithm.

As each bin can contain items of volume at most 1, it is easy to see that the d-dimensional Vector Packing problem is space related.

Remark 4. The d-dimensional Vector Packing problem is space related.

We will now present a flexible online algorithm with ratio $\beta = 2d$ that is a simple adaption of the well-known next fit online algorithm for bin packing. Every bin will have an index to guarantee a linear ordering. Whenever a vector w arrives, we first check whether w can be packed into an existing bin. If this is possible, we add w to such a bin with minimal index. If no such bin exists, we open a new bin containing w. If we are given a previous packing S, we simply ignore the previous bins and do not put any vector in them.

Theorem 2.6.1. *For every $d \geq 1$, the presented algorithm A_{VP} is a flexible online algorithm for d-dimensional Vector Packing with ratio $2d$.*

Proof. The flexibility of A_{VP} follows directly due to the fact that we only build upon the existing packing. We will now show that A_{VP} has ratio $2d$. Let t be any time point, S be the previous packing we built upon, and $B_1, \dots, B_k$ be the bins opened by A_{VP}. Let J be the set of vectors packed into $B_1, \dots, B_k$. The current packing then uses $|S| + k$ bins, where $|S|$ denotes the number of bins used by the previous packing S. Consider two adjacent bins B_i and B_{i+1} for $i \in \{1, \dots, k-1\}$. We have opened bin B_{i+1}, because there is some vector $w' \in B_{i+1}$ and some index $j \in \{1, \dots, d\}$ such that $\left[\sum_{w \in B_i} w[j]\right] + w'[j] > 1$. Denoting this index with $j = j(i)$, we have $\sum_{w \in B_i \cup B_{i+1}} w[j(i)] > 1$ and thus $d \cdot \textsc{vol}(B_i \cup B_{i+1}) > 1$. Hence, we can conclude

$$k < 1 + \sum_{i=1}^{k-1} d \cdot \textsc{vol}(B_i \cup B_{i+1}) = 1 + d \cdot \sum_{i=1}^{k-1} (\textsc{vol}(B_i) + \textsc{vol}(B_{i+1})) =$$

$$1 + d \cdot \left[(\sum_{i=1}^{k-1} \textsc{vol}(B_i)) + (\sum_{i=2}^{k} \textsc{vol}(B_i)) \right] < 1 + 2d(\sum_{i=1}^{k} \textsc{vol}(B_i)) = 1 + 2d\,\textsc{vol}(J). \square$$

We have now shown the first two ingredients for our framework: the problem is space related and we gave a suitable online algorithm. The final piece – an offline approximation algorithm – is given by the algorithm of Bansal *et. al.* [6] which is a $\ln(d+1) + 0.807 + \epsilon$-approximation. We can thus use Theorem 2.2.1 with $\gamma = \ln(d+1) + 0.807 + \epsilon$ and $\beta = 2d$ to conclude the following theorem.

Theorem 2.6.2. *For every $d \geq 1$, there is a robust online algorithm for the dynamic d-dimensional Vector Packing problem that is $\ln(d+1) + 0.807 + \epsilon$-competitive and has amortized migration factor $O(1/\epsilon)$.*

Chapter 3

Robust Online Algorithms for Maximization Problems

Semi-online algorithms that are allowed to perform a bounded amount of repacking achieve guaranteed good worst-case behaviour in a more realistic setting. Most of the previous works focused on minimization problems that aim to minimize some costs. In this chapter, we study maximization problems that aim to maximize their profit.

We mostly focus on a class of problems that we call *choosing problems*, where a maximum profit subset of a set objects has to be maintained. Many known problems, such as KNAPSACK, MAXIMUMINDEPENDENTSET and variations of these, are part of this class. We present a framework for choosing problems that allows us to transfer offline α-approximation algorithms into $(\alpha - \epsilon)$-competitive semi-online algorithms with amortized migration $O(1/\epsilon)$. Moreover we complement these positive results with lower bounds that show that our results are tight in the sense that no amortized migration of $o(1/\epsilon)$ is possible.

3.1 Introduction

Optimization problems and how fast we can solve them optimally or approximatively have been a central topic in theoretical computer science. These kind of problems usually have their origins in the real world and solving them in most cases is not only relevant from a theoretical perspective but also has many applications. These optimization problems do not account for one major problem that is unique to applications: an unknown future. Usually, we are not given all the information in advance, as unforeseeable things like customers cancelling or new urgent customer requests can happen at any moment. This context gave rise to the study of *online problems* in different variants to model this uncertainty. The classical model starts with an empty instance and in subsequent time steps, new parts of the instance are added. In order to solve a problem, an online algorithm must generate a solution for every time step without knowing any information about future events.

In the strictest setting, the algorithm is not allowed to alter the solution generated in a previous step at all, so every mistake will carry weight into the future. As this is a very heavy restriction for online algorithms, there are also variants where the algorithm is allowed to change solutions to some degree. We cannot allow for an arbitrary number of changes as this only leads to the offline setting. We therefore consider the *migration model*, where every change in the instance, e. g. an added node to a graph or a new item, comes with a *migration potential*. Intuitively, this migration potential is linked to some size or weight which means objects that have a larger impact on the optimization criteria will yield larger migration potential allowing more

change. Similarly, small objects will only allow for small changes of the solution. The ratio between the sum of changed objects in the solution and the total migration potential at a time is the so called *migration factor*.

We consider this migration setting in an amortized way by allowing the migration to accumulate over time. This way we allow our algorithm to generally handle newly arriving objects without changing the solution (apart from extending the solution with regards to the new objects) and at some later point of time we will repack the solution and use the sum of all migration potential of items that arrived up to that time. On the matter of (amortized) migration there are two criteria to consider. For one, as usual in optimization theory, we want to achieve good competitive ratios, meaning a solution close to an optimal offline solution. On the other hand, we would like that the migration also remains small. In general this is quite an intricate question as both these factors counteract each other. The better the solution we generate, the more we need to repack and vice versa. Despite this duality, we will present in this section a very simple framework that achieves results close to the best offline results for a large range of problems. In fact, we manage to keep solutions on par with the best offline algorithms except an additive ϵ-term. Surprisingly, we only need an amortized migration factor of $O(1/\epsilon)$ despite maintaining such a high quality solution. For many problems this framework even works when considering the problem variants where objects not only appear but are also removed from the instance. In addition to these positive results, we also show that a migration of only $\Omega(1/\epsilon)$ is needed even for relatively simple problems such as the SUBSETSUM problem.

3.2 Preliminaries

We are given some optimization problem Π_{off} consisting of a set of objects and consider the online version Π_{on}, where these objects arrive one by one over time (the *static* case). If arrived items can also be removed from the instance, we call this the *dynamic* case. For an instance $I \in \Pi_{\text{on}}$ and some time t we denote the instance at time t containing the first t objects by I_t. As discussed above, we allow a certain amount of repacking and thus, every object has an associated *migration potential*, which typically corresponds to its size or its weight.

The item arriving or departing at time t has migration potential $\Delta(I_t)$. Generally in the following, we will write $\Delta(I_t \to I_{t'})$ to denote the migration potential that we received starting at time t until t' for $t \leq t'$. If the given instance is clear from the context, we will simplify this notation and write $\Delta_{t:t'} := \Delta(I_t \to I_{t'})$. The total migration potential up to some time t is thus given by $\Delta_{0:t}$.

We further assume that for every two feasible solutions S_t and $S'_{t'}$ at times t, t', we also have a necessary migration cost that we denote by $\phi(S_t \to S'_{t'})$, respectively. This resembles the costs to migrate from solution S_t to solution S'_t. We assume that these costs can be computed easily by comparing the respective solutions. Note that the initial assignment of an item does not cost any migration. Hence, we often write $\phi(S_t \to S_{t+1})$ to denote the migration cost of changing solution S_t to solution S_{t+1} and assume that the newly arrived item is now also present in S_{t+1}.

We say that an online algorithm has *amortized migration factor* γ if, for all time steps t, the sum of the migration costs is at most $\gamma \cdot \sum_{i=1}^{t} \Delta(I_t) = \gamma \cdot \Delta_{0:t}$, i. e. $\sum_{i=1}^{t} \phi(S_{i-1} \to S_i) \leq \gamma \cdot \Delta_{0:t}$, where S_i are the solutions produced by the algorithm with S_0 being the empty solution. We will sometimes make use of the amortized migration factor inside a time interval $t \to \cdots \to t'$, which we will denote by $\gamma_{t:t'}$. The notions of

migration or *migration factor* are interchangeably used by us and always describe the amortized migration factor.

In this section, we only consider maximization problems. Hence, every solution S_t of an instance I_t has some profit $\text{PROFIT}(S_t)$ and $\text{OPT}(I_t)$ denotes the optimal profit of any solution to I_t. An algorithm that achieves competitive ratio $1-\epsilon$ and has migration $f(1/\epsilon)$ for some function f is called *robust* [78].

3.2.1 KNAPSACK-type problems

The KNAPSACK problem is one of the classical maximization problems. In its most basic form, it considers a *capacity* $C \in \mathbb{N}$ and a finite set I of *items*, each of which is assigned a *weight* $w_i \in \mathbb{N}$ and a *profit* $p_i \in \mathbb{N}$. The objective is to find a subset $S \subseteq I$, interpreted as a packing of the figurative knapsack, with maximum profit $\text{PROFIT}(S) = \sum_{i \in S} p_i$ while the total weight $\text{WEIGHT}(S) = \sum_{i \in S} w_i$ does not exceed C. The special case in which all weights are equal to their respective profits is the SUBSETSUM problem. In this case, because weight and profit coincide, we will simply call both the *size* of an item. A natural generalization of KNAPSACK is to generalize capacity and weight vector to be d-dimensional vectors, i.e. $C, w_i \in \mathbb{N}^d$ for some $d \in \mathbb{N}$. The problem of finding a maximum profit packing fulfilling all d constraints is then known as the d-dimensional KNAPSACK problem.

In the MULTIPLEKNAPSACK problem, one is given an instance I consisting of items with assigned weights and profits, just like in the KNAPSACK problem. However, in contrast, we are given not one but m different knapsacks with respective capacities $C^{(1)}, ..., C^{(m)}$. The goal is to find m disjoint subsets $S^{(1)}, ..., S^{(m)}$ of I, such that the total profit $\sum_{j=1}^{m} \text{PROFIT}(S^{(j)})$ is maximized w.r.t. the capacity conditions

$$\text{WEIGHT}(S^{(j)}) \leq C^{(j)}, \quad \forall j = 1, ..., m.$$

Note that the KNAPSACK problem is a special case of the MULTIPLEKNAPSACK problem with $m = 1$.

Another generalization of the standard KNAPSACK problem is 2DGEOKNAPSACK. This problem takes as input the width $W \in \mathbb{N}$ and height $H \in \mathbb{N}$ of the knapsack and a set I of axis-aligned rectangles $r \in I$ with widths $w_r \in \mathbb{N}$, heights $h_r \in \mathbb{N}$, and profits $p_r \in \mathbb{N}$. An optimal solution to this instance consists of a subset $S \subseteq I$ of the rectangles together with a non-overlapping axis-aligned packing of S inside the rectangular knapsack of size $W \times H$ such that $\text{PROFIT}(S)$ is maximized.

3.2.2 Independent Set

Another classical optimization problem is the MAXIMUMINDEPENDENTSET problem. While it can be considered for different types of graphs, the most basic variant is defined on a graph $G = (V, E)$ with a set of nodes V and a set of corresponding edges E. A subset $S \subseteq V$ is called *independent* if for all $u, v \in S$ it holds that $(u, v) \notin E$, i.e. u and v are not neighbours. A maximal independent set is then an independent set that is no strict subset of another independent set. MAXIMUMINDEPENDENTSET is the problem of finding a maximum independent set for a given graph G. In the literature, MAXIMUMINDEPENDENTSET is, among others, studied in planar, perfect, or claw-free graphs. For the online variant, we usually assume that a node is added (or removed) to the instance in every time step along with its adjacent edges.

Closely related to the well-studied MAXIMUMINDEPENDENTSET problem is the MAXIMUMDISJOINTSET problem. For a given instance I that consists of items with

a geometrical shape, the goal is to find a largest disjoint set which is a set of non-overlapping items. As we can convert an MDS instance to an MIS instance, we sometimes use MIS to also denote this problem. MDS is often considered limited to certain types of objects. These can be (unit-sized) disks, rectangles, polygons or other objects, and any d-dimensional generalization of them. We will also consider pseudo-disks, which are objects that pairwise intersect at most twice in an instance.

The standard MIS and MDS problems are both special cases of the generalized MAXWEIGHTINDEPENDENTSET or MAXWEIGHTDISJOINTSET, respectively, where each node i is assigned a profit value w_i. The goal for these problems is to find a maximum profit independent subset.

3.2.3 Our Results

Our main result is a framework that is strongly inspired by the approach of Berndt *et al.* [15]. For minimization problems, they proposed a framework using two known algorithms, one online and one offline algorithm, in order to solve a given problem. This approach behaves a bit differently for maximization problems in terms of the competitive ratio. While a respective α-approximation offline algorithm paired with a fitting β-competitive online algorithm yields an $\alpha + O(1)\beta\epsilon$ competitive-algorithm for minimization problems, we show that for maximization problems such fitting algorithms will result in a $\alpha \cdot \beta$-competitive algorithm. Since we do not focus on this result more and do not give an explicit application for this version the analysis on this framework can be found in section 3.6.

We discuss a class of problems that is characterized by the common task of choosing a subset of objects with maximum profit fulfilling some secondary constraints. Many important problems like the above presented variants of KNAPSACK or MAXIMUMINDEPENDENTSET are covered by this class of problems. We show that for these kind of problems, which we will call *choosing problems* in the following, the framework can be simplified by completely removing the online algorithm.

Using a simplified framework, we achieve $(1-\epsilon)$-competitive algorithms for KNAPSACK even when generalized to arbitrary but fixed dimension d. In the 2DGEOKNAPSACK where we additionally interpret items as rectangles that need to be packed into a rectangular knapsack, we achieve a $(\frac{9}{17}-\epsilon)$-competitive ratio. We also consider problem variants outside of the class of choosing problems and show that the static cases of MAXIMUMINDEPENDENTSET for planar graphs with arriving edges and MULTIPLEKNAPSACK also admit robust approximation schemes. We complement these positive results by also proving lower bounds for the necessary migration showing that some of our results are indeed tight. We also give lower bounds for different variants, including starting with an adversarially chosen solution and different migration models.

PROBLEM	Competitive Ratio	MF	lower bound MF
KNAPSACK/SUBSETSUM (fixed d)	$(1-\epsilon)$	$O(1/\epsilon)$	$\Omega(1/\epsilon)$
2DGEOKNAPSACK	$(9/17-\epsilon)$	$O(1/\epsilon)$	open [1]
MIS (unweighted planar)	$(1-\epsilon)$	$O(1/\epsilon)$	open
MIS (unweighted unit-disk)	$(1-\epsilon)$	$O(1/\epsilon)$	open
MIS (weighted disk-like objects[2])	$(1-\epsilon)$	$O(1/\epsilon)$	$\Omega(1/\epsilon)$
MIS on pseudo-disks	$(1-\epsilon)$	$O(1/\epsilon)$	open
MULTIPLEKNAPSACK	$(1-\epsilon)$	$O(1/\epsilon)$	$\Omega(1/\epsilon)$
MIS (unw. planar, edge arrival)	$(1-\epsilon)$	$O(1/\epsilon)$	open

[1] We prove a lower bound of $\Omega(1/\epsilon)$ for comp. ratio $(1-\epsilon)$.

[2] Result applies to all objects that the PTAS from Erlebach *et al.* [46] can be used on, even in higher dimensions.

3.2.4 Related Work

Upper Bounds: The general idea of bounded migration was introduced by Sanders, Sivadasan, and Skutella [78]. They developed an $(1+\epsilon)$-competitive algorithm with non-amortized migration factor $f(1/\epsilon)$ for the MAKESPANSCHEDULING problem. Gálvez *et al.* [48] showed two $(c+\epsilon)$-competitive algorithms with migration factor $(1/\epsilon)^{O(1)}$ for some constants c for the same problem. Skutella and Verschae [79] were able to transfer the $(1+\epsilon)$-competitive algorithm also to the setting, where items depart. They also considered the MACHINECOVERING problem and obtained an $(1+\epsilon)$-competitive algorithm with amortized migration factor $f(1/\epsilon)$. For BINPACKING, Epstein and Levin [42] presented a $(1+\epsilon)$-competitive algorithm with non-amortized migration factor $f(1/\epsilon)$. Jansen and Klein [62] were able to obtain a non-amortized migration factor of $(1/\epsilon)^{O(1)}$ for this problem, and Berndt *et al.* [12] showed that such a non-amortized migration factor is also possible for the scenario, where items can depart. Considering amortized migration, Feldkord *et al.* [47] presented a $(1+\epsilon)$-competitive algorithm with migration factor $O(1/\epsilon)$ that also works for departing items. Epstein and Levin [44] investigated a multidimensional extension of BINPACKING problem, called HYPERCUBEPACKING where hypercubes are packed geometrically. They obtained an $(1+\epsilon)$-competitive algorithm with worst-case migration factor $f(1/\epsilon)$. For the preemptive variant of MAKESPANSCHEDULING, Epstein and Levin [43] obtained an exact online algorithm with non-amortized migration factor $1+1/m$. Berndt *et al.* [13] studied the BINCOVERING problem with amortized migration factor and non-amortized migration factor and obtained $(1+\epsilon)$-competitive algorithms and matching lower bounds, even if items can depart. Jansen *et al.* [65] developed a $(1+\epsilon)$-competitive algorithm with amortized migration factor $f(1/\epsilon)$ for the STRIPPACKING problem. Finally, Berndt *et al.* [15] developed a framework similar to this work, but only for *minimization problems*, and showed that for a certain class of packing problems, any c-approximate algorithm can be combined with a suitable online algorithm to obtain a $(c+\epsilon)$-competitive algorithm with amortized migration factor $O(1/\epsilon)$.

Lower Bounds: Skutella and Verschae [79] showed that a non-amortized migration factor of $f(1/\epsilon)$ is not possible for any function f for MACHINECOVERING. Berndt *et al.* showed that a non-amortized migration factor of $\Omega(1/\epsilon)$ is needed [12] for BINPACKING, and Feldkord *et al.* [47] showed that this also holds for the amortized migration factor. Epstein and Levin [43] showed that exact algorithms for the makespan minimization problem on uniform machines and for identical machines in the restricted assignment setting have worst-case migration factor at least $\Omega(m)$.

Dynamic Algorithms: In the semi-online setting, there are several metrics that one tries to optimize. First of all, there is the competitive ratio measuring the quality of the solution. Second, in order to prevent that the online problem simply degrades to the offline problem, one needs to bound some resource. In the setting that we consider, we bound the amount of repacking possible, as such a repacking often comes with a high cost in practical applications. In an alternate approach, often called *dynamic algorithms*, we restrict the running time needed to update a solution, ideally to a sub-linear function (see e. g. the surveys [19, 57]). Note that the amount of repacking used here can be arbitrarily high (using a suitable representation of the current solution). This setting has been also studied recently for KNAPSACK variants [40] and MAXIMUMINDEPENDENTSET variants [58]. There are also works aiming to combine both of the before mentioned approaches [73, 55].

3.3 Upper Bounds for Choosing Problems

3.3.1 Framework for Choosing Problems

In this section, we will consider the aforementioned class of *choosing problems*. In general, a choosing problem is defined by a set of objects with some properties, and the objective is to select a subset of these objects with maximum profit, while potentially respecting some secondary constraints.

Definition 3.3.1. Consider a problem Π where every instance $I \in \Pi$ is given by a set of objects, where each object $i \in I$ is assigned a fixed profit value p_i, and a set of feasible solutions $\text{SOL}(I)$. We call Π a choosing problem if $\text{SOL}(I) \subseteq \mathcal{P}(I)$, and the objective of some instance $I \in \Pi$ is to find a subset $S \in \text{SOL}(I)$ while maximizing the total profit $\text{PROFIT}(S) = \sum_{i \in S} p_i$. We further make the following two requirements for choosing problems that we will discuss in this part:

(i) For any feasible solution $S \in \text{SOL}(I)$, we have that any subset $S' \subseteq S$ is also a feasible solution, i. e. $S' \in \text{SOL}(I)$.

(ii) For any solution $S \in \text{SOL}(I_t)$ for an instance I_t with respective follow-up instance $I_{t+1} = I_t \triangle \{i_{t+1}\}$, the solution $S' := S \backslash \{i_{t+1}\}$ stays feasible for I_{t+1}, i. e. $S' \in \text{SOL}(I_{t+1})$.

For choosing problems, the profits of objects are their migration potential and costs; $\Delta(I_t) = p_i$, where i is the object added or removed at time t. Given two solutions $S_1, S_2 \subseteq I$, we further have that $\phi(S_1 \to S_2) = \text{PROFIT}(S_1 \triangle S_2)$.

While we restrict the range of problems with these properties, it is necessary to do so since an adversary can enforce an unreasonably high migration factor for problems we excluded this way. In particular the first property guarantees that there is no low profit item with low migration potential added by the adversary which would allow for a completely new solution with high profit that we would need to switch to. The second property serves a similar purpose, as it prevents the adversary from adding any arbitrary items making our current solution infeasible.

We could approach choosing problems like Berndt *et al.* [15] and use a greedy online algorithm in conjunction with the best known offline algorithm. While this would also create good results, we want to show that an online algorithm is not even necessary. We propose instead the algorithm that computes an offline solution S with profit V and then waits until the total profit of items being added or removed from the instance exceeds ϵV. At this point we simply replace S with a new offline solution for the current instance. This algorithm already achieves a competitive rate close to the approximation ratio of the offline algorithm.

Theorem 3.3.1. *Let A_{off} be an offline algorithm with an approximation ratio of α. Then the resulting framework using A_{off} is a $(1-\epsilon) \cdot \alpha$-competitive algorithm requiring a migration factor of $O(1/(\epsilon \cdot \alpha))$.*

Proof of Theorem 3.3.1. In the following, we refer to time steps where the offline algorithm is applied as repacking times. Note that for the start of any online instance and the first arriving items, we will just add the first eligible item with the offline algorithm when it arrives. This results in the first repacking time. Consider now any repacking time t with generated solution S_t, and let t' be the first point of time where $\Delta_{t:t'} > \epsilon \, \text{PROFIT}(S_t)$. First we will show that the migration factor of the applied framework is small. We do so by proving that the migration potential between

two repacking times accommodates for the repacking at the later repacking time. Denote with $A_{t:t'}$ the total profit of items that arrived up till time t'.

We now have that

$$\frac{\phi(S_t \to S_{t'})}{\Delta_{t:t'}} \leq \frac{\text{OPT}(I_t) + \text{OPT}(I_{t'})}{\Delta_{t:t'}} \leq \frac{\text{OPT}(I_t) + \text{OPT}(I_t) + A_{t:t'}}{\Delta_{t:t'}} \leq$$
$$\frac{\text{OPT}(I_t) + \text{OPT}(I_t) + \Delta_{t:t'}}{\Delta_{t:t'}} \leq \frac{2 \cdot \text{OPT}(I_t)}{\Delta_{t:t'}} + 1 \leq \frac{2 \cdot \text{PROFIT}(S_t)}{\alpha \Delta_{t:t'}} + 1 <$$
$$\frac{2 \cdot \text{PROFIT}(S_t)}{\alpha\epsilon \,\text{PROFIT}(S_t)} + 1 \in O(1/(\epsilon \cdot \alpha)).$$

Now it is left to show that up till time $t' - 1$ we have maintained $(1-\epsilon)\alpha$ competitive rate. Denote with $A_{t:t'-1}$ and $R_{t:t'-1}$ the total profit of items that arrived or departed up till time $t' - 1$, and note that by choice of t', we have that $A_{t:t'-1} + R_{t:t'-1} \leq \epsilon \,\text{PROFIT}(S_t)$. We now have

$$\text{PROFIT}(S_{t'-1}) \geq \text{PROFIT}(S_t) - R_{t:t'-1} =$$
$$\text{PROFIT}(S_t) + A_{t:t'-1} - A_{t:t'-1} - R_{t:t'-1} \geq \text{PROFIT}(S_t) + A_{t:t'-1} - \epsilon \,\text{PROFIT}(S_t) =$$
$$(1-\epsilon)\,\text{PROFIT}(S_t) + A_{t:t'-1} \geq (1-\epsilon)\alpha \,\text{OPT}(S_t) + A_{t:t'-1} \geq$$
$$(1-\epsilon)\alpha(\text{OPT}(S_t) + A_{t:t'-1}) \geq (1-\epsilon)\alpha(\text{OPT}(S_{t'-1})).$$

□

3.3.2 Resulting upper bounds and necessary migration

The framework introduced can be applied to any choosing problem with some existing offline algorithm. We observe that by using Theorem 3.3.1 for problems that admit a PTAS, we get a respective robust PTAS, and for other problems, we achieve the ratio of any offline algorithm with small additional error. We note without proof that all problems mentioned in the following theorem are choosing problems as solutions are made up of sets of items or nodes and they stay feasible under the required circumstances. While we give a general proof here, in the following section we will further discuss why each problem is a choosing problem.

Theorem 3.3.2. *For the following problems there exists an online algorithm with competitive ratio* $1 - \epsilon$ *and migration factor* $O(1/\epsilon)$.

- SUBSETSUM *and* KNAPSACK *using the FPTAS by Jin [67]*
- *d-dimensional* KNAPSACK *using the PTAS from Caprara* et al. *[25]*
- MAXIMUMINDEPENDENTSET *on unweighted planar graphs by using the PTAS by Baker [3]*
- MAXIMUMINDEPENDENTSET *on (weighted) d-dimensional disk-like objects with fixed d using the PTAS by Erlebach* et al. *[46]*
- MAXIMUMINDEPENDENTSET *on pseudo-disks using a PTAS by Chan and Har-Peled [27]*

Additionally, for the 2DGEOKNAPSACK *problem, there exists an online algorithm with a competitive ratio* $9/17 - \epsilon$ *and migration factor* $O(1/\epsilon)$ *by using the* $9/17$ *approximation algorithm from Gálvez* et al. *[49]*

Proof of Theorem 3.3.2. We prove that a $(1-\epsilon)$ approximation yields the desired result. The statement for 2DGEOKNAPSACK follows similarly. Note that using a $(1-\epsilon)$ approximation with Theorem 3.3.1, we achieve an online algorithm with a competitive ratio of $(1-\epsilon)^2 = 1-2\epsilon+\epsilon^2$. By applying the framework with $\epsilon' = 1/2\epsilon$, we achieve the desired PTAS quality. The required migration of our framework is bounded by $O(\frac{1}{\epsilon(1-\epsilon)}) = O(1/\epsilon)$. □

We also show how to apply our framework in a setting outside of choosing problems. We consider two problems still similar to the given context of KNAPSACK and MAXIMUMINDEPENDENTSET.

Theorem 3.3.3. *The static cases of* MULTIPLEKNAPSACK *and* MAXIMUMINDEPENDENTSET *on weighted planar graphs when adding edges both admit a robust PTAS with competitive ratio* $1-\epsilon$ *and migration factor* $O(1/\epsilon)$.

Finally, one might ask how much migration is actually needed to achieve such results. One can come up with quite simple lower bounds on the necessary migration for these kind of problems by forcing the algorithm to switch between two different solutions. If the adversary accomplishes this using little migration but with at least one solution being reasonably expensive to replace, then this yields an interesting lower bound for SUBSETSUM and MAXIMUMINDEPENDENTSET showing that our framework is indeed tight in migration for all KNAPSACK variants and some MAXIMUMINDEPENDENTSET variants.

Theorem 3.3.4. *For the online* SUBSETSUM *problem and (weighted)* MAXIMUMINDEPENDENTSET *problem, there is an instance such that the migration needed for a solution with value* $(1-\epsilon)\,\mathrm{OPT}$ *is* $\Omega(1/\epsilon)$.

3.3.3 Knapsack Problems

We will now discuss the problems in further detail. Note that KNAPSACK and the optimization variant of SUBSETSUM trivially are part of the class of choosing problems. The goal of these problems is to find a feasible subset, that is feasible in regards to the capacity constraint of the given Knapsack. Observe that both additional requirements for choosing problems are also fulfilled: Removing any item from a feasible solution will not make it infeasible as the new solution will have less total weight and in the same way adding some item i will not make any prior infeasible subset of items feasible as it will only increase the total weight. Since KNAPSACK is a well studied problem we have access to a wide range of algorithms. Since we are looking to achieve a robust PTAS result, we will apply the FPTAS of Jin [67] and obtain the following result with our framework.

Theorem 3.3.5. *For the* KNAPSACK *and accordingly* SUBSETSUM *problem, there exists an online algorithm with competitive ratio* $1-\epsilon$ *and migration factor* $O(1/\epsilon)$.

Proof. By applying the given FPTAS with Theorem 3.3.1 we achieve an online algorithm with a competitive rate of $(1-\epsilon)^2 = 1-2\epsilon+\epsilon^2$. We can therefore apply the framework with $\epsilon' = 1/2\epsilon$ yielding the desired solution quality. The required migration of our framework is bounded by $O(\frac{1}{\epsilon(1-\epsilon)}) = O(1/\epsilon)$. □

The d-dimensional KNAPSACK trivially falls in the class of choosing problems similar to its special case for $d = 1$. For this problem, we use the PTAS from Caprara *et al.* [25]. As with the previous result we gain a similar result.

Theorem 3.3.6. *For any fixed $d \in \mathbb{N}$, for the d-dimensional* KNAPSACK *problem there exists an online algorithm with competitive ratio $1 - \epsilon$ and migration factor $O(1/\epsilon)$.*

The geometric variants of, for example 2DGEOKNAPSACK, add another layer to the problem by packing geometric objects into some actual knapsack. In our framework however, this task is simply delegated to the respective offline algorithm. Therefore we can use the $\frac{9}{17} - \epsilon$ approximative algorithm from Gálvez *et al.* [49] for the two-dimensional version of this problem that we denoted as 2DGEOKNAPSACK.

Theorem 3.3.7. *For the* 2DGEOKNAPSACK *problem there exists an online algorithm with a competitive ratio $9/17 - \epsilon$ and migration factor $O(1/\epsilon)$.*

3.3.4 Maximum independent set problems with incoming nodes

As for MAXIMUMINDEPENDENTSET being also regarded as a choosing problem, we can firstly confirm that the properties from Section 3.3.1 hold. That is, if we consider a problem instance I and an independent set $S \subseteq I$, then every subset $S' \subseteq S$ is still a set of non-adjacent nodes and therefore remains an independent set. Furthermore, a solution S to the instance I_t stays feasible for the next instance I_{t+1} since no new edges connecting already present nodes are added. In the following, we describe how our framework can be applied to some variants of MAXIMUMINDEPENDENTSET.

Generally, when speaking about the unweighted variants, we assume the profit of every node (or item) in the instance to be 1.

Maximum independent set for planar graphs While the maximum independent set is difficult to approximate [1], we limit ourselves to certain graph classes. This gives us the opportunity again to apply our framework on some efficient offline algorithms. One graph class that is studied well in this context is the class of planar graphs, that contains graphs that can be drawn on the plane without edge-intersections. For MAXIMUMINDEPENDENTSET in planar graphs, we will use the PTAS by Baker [3] to achieve a robust PTAS similar to KNAPSACK.

Theorem 3.3.8. *For the* MAXIMUMINDEPENDENTSET*, there exists an online algorithm with competitive ratio $1 - \epsilon$ and migration factor $O(1/\epsilon)$.*

Maximum disjoint set for unit disks An instance for the MAXIMUMDISJOINTSET problem can be easily transferred into a MAXIMUMINDEPENDENTSET instance by replacing every disk by a node and join two nodes by an edge if the corresponding disks are overlapping. Also MAXIMUMDISJOINTSET fulfils the aforementioned properties. That is, if we consider a disjoint set $S \subseteq I$, then every subset $S' \subseteq S$ is still a set of non-overlapping items and therefore a disjoint set. Furthermore a solution S to an instance I_t stays feasible for the next instance I_{t+1} since the position of items in S is not changing and therefore remains a disjoint set.

For the offline algorithm solving MAXIMUMDISJOINTSET, we follow the approach of Erlebach, Jansen and Seidel [46] which gives us an PTAS for the offline (weighted) MAXIMUMINDEPENDENTSET in disk graphs.

Theorem 3.3.9. *For the* MAXIMUMINDEPENDENTSET *problem on (weighted) unit-disk graphs there exists a robust PTAS.*

LISTING 3.1: Online algorithm $A_{\text{on}}^{\text{MK}}$ for MULTIPLEKNAPSACK

```
compute a packing S_1 for the first instance I_1 using A_off;
set Δ := 0 and P := PROFIT(S_1);
for each time step t−1 → t with t > 1 and new item i:
  set S_t := S_{t−1};
  increment Δ by the profit p_i of i;
  if (†) Δ > εP:
    do a repacking with A_off to obtain the new solution S_t;
    set Δ := 0 and P := PROFIT(S_t);
  endif;
```

MaximumDisjointSet for pseudo-disks For the MAXIMUMDISJOINTSET problem limited to pseudo-disks, we have another PTAS result in the offline world introduced by Chan and Har-Peled [27]. Using their PTAS, we get yet another robust PTAS in the dynamic online world.

Theorem 3.3.10. *For the* MAXIMUMDISJOINTSET *problem on pseudo-disks there exists a robust PTAS.*

3.4 Upper bounds for non-choosing problems

There are many maximization problems that do not fit neatly under the umbrella of choosing problems. However, it turns out that there is a wider class of problems that allow a similar treatment. In this section, we examine two problems that are not choosing problems (at least as defined here): MULTIPLEKNAPSACK and MAXIMUMINDEPENDENTSET in weighted planar graphs with new edges added in each time step. The first problem is not a choosing problem essentially because it requires the selection of *multiple* subsets of a given instance, not just one. The second problem is not a choosing problem because each time steps adds a new *edge* to the graph instead of a *vertex*.

3.4.1 MULTIPLEKNAPSACK

It is known that the MULTIPLEKNAPSACK problem has an offline PTAS [61, 60, 29, 69], which we will denote by A_{off} in this subsection. On the basis of this PTAS, given ϵ, we define the polynomial-time online algorithm $A_{\text{on}}^{\text{MK}}$ above.

Theorem 3.4.1. *The algorithm $A_{\text{on}}^{\text{MK}}$ computes $(1-\epsilon)$-approximate solutions.*

Proof. Whenever a repacking took place, the resulting solutions are $(1-\epsilon)$-approximate. If t' is any time point at which no repacking took place, let t be the latest time point before t' at which a repacking did take place. In this case, we see that

$$\begin{aligned}
&A_{\text{on}}^{\text{MK}}(I_{t'}) = \text{PROFIT}(I_t) \geq (1-\epsilon)\,\text{OPT}(I_t) \geq (1-\epsilon)(\text{OPT}(I_{t'}) - \Delta_{t:t'}) \geq \\
&(1-\epsilon)(\text{OPT}(I_{t'}) - \epsilon \cdot \text{PROFIT}(I_{t'})) \geq (1-\epsilon)(\text{OPT}(I_{t'}) - \epsilon \cdot \text{OPT}(I_{t'})) \geq \\
&(1-\epsilon)^2\,\text{OPT}(I_{t'}) \geq (1-2\epsilon)\,\text{OPT}(I_{t'}).
\end{aligned}$$

Thus, we see that, after a linear reparametrization of ϵ, the algorithm $A_{\text{on}}^{\text{MK}}$ computes $(1-\epsilon)$-approximate solutions. □

Theorem 3.4.2. *The migration factor of $A_{\text{on}}^{\text{MK}}$ is bounded by $O(1/\epsilon)$.*

LISTING 3.2: Online algorithm $A^{\text{IS}}_{\text{on}}$ for MAXIMUMINDEPENDENTSET in planar graphs

```
compute an IS S_1 for the first graph G_1 using A_off;
set Δ := 0 and W := WEIGHT(S_1);
for each time step t−1 → t with t > 1 and new edge e:
  if e has at most one adjacent vertex in S_{t−1}:
    set S_t := S_{t−1};
  else:
    remove the smaller of the two adjacent vertices of e
    from S_{t−1} to obtain S_t;
  endif;
  increment Δ by the minimum of the weights of the two
  adjacent vertices of e;
  if (†) Δ > εW:
    use A_off to obtain the new solution S_t;
    set Δ := 0 and W := WEIGHT(S_t);
  endif;
```

Proof. It suffices to show the bound for individual phases in order to show the bound for all time intervals. Let $t \to \cdots \to t'$ be a phase. This implies that no repackings took place except at $t-1 \to t$ and $t'-1 \to t'$. Then

$$\gamma_{t:t'} \leq \frac{\text{OPT}(I_{t'})}{\Delta_{t:t'}} \leq \frac{\text{OPT}(I_t) + \Delta_{t:t'}}{\Delta_{t:t'}} = \frac{\text{OPT}(I_t)}{\Delta_{t:t'}} + 1 \leq$$
$$\frac{\text{OPT}(I_t)}{\epsilon \,\text{PROFIT}(I_{t'})} + 1 \leq \frac{\text{OPT}(I_t)}{\epsilon(1-\epsilon)\,\text{OPT}(I_t)} + 1 = O(1/\epsilon),$$

which proves the upper bound. □

3.4.2 Maximum independent set in weighted planar graphs with edge arrivals

We are given a weighted planar graph $G = (V, E)$ and a sequence of subgraphs $(G_t)_{t=1}^T$ with $G_t = (V, E_t)$, for $E_t \subseteq E$. Moreover, we assume that $|E_t \setminus E_{t-1}| = 1$ for all $t = 2, ..., T$, i.e. every time step adds exactly one edge to the graph. We interpret this sequence as an online instance of the static case of MAXIMUMINDEPENDENTSET. Given $\epsilon \in (0, \frac{1}{2})$, we aim to find a polynomial-time online algorithm that computes $(1-\epsilon)$-approximate solutions with a migration factor $\mathcal{O}(1/\epsilon)$, where the migration potential of an edge is given by the minimum of the weights of the two adjacent vertices. Note that every graph G_t in this sequence is planar, hence we know that there is an offline PTAS A_{off} that can find $(1-\epsilon)$-approximate independent sets (IS) in polynomial time [3]. Using this, we define the online algorithm $A^{\text{IS}}_{\text{on}}$ above.

Theorem 3.4.3. *The algorithm $A^{\text{IS}}_{\text{on}}$ computes $(1-\epsilon)$-approximate solutions.*

Proof. Clearly, the approximation ratio is satisfied whenever a repacking took place. Thus, we let t' be a time at which no repacking took place and t the latest time before t' where a repacking did take place. We see that

$$A^{\text{IS}}_{\text{on}}(G_{t'}) = \text{WEIGHT}(S_{t'}) \geq \text{WEIGHT}(S_t) - \Delta_{t:t'} \geq_{(\dagger)} (1-\epsilon)\,\text{WEIGHT}(S_t) \geq$$
$$(1-\epsilon)^2\,\text{OPT}(G_t) \geq (1-\epsilon)^2\,\text{OPT}(G_{t'}) \geq (1-2\epsilon)\,\text{OPT}(G_{t'}).$$

Thus, we have proven that, after a linear reparametrization of ϵ, the algorithm $A^{\text{IS}}_{\text{on}}$ computes $(1-\epsilon)$-approximate solutions. □

Theorem 3.4.4. *The migration factor of $A^{\text{IS}}_{\text{on}}$ is bounded by $O(1/\epsilon)$.*

Proof. To show the upper bound on the total migration factor, it suffices to show the upper bound for individual phases. To that end, let $t \to \cdots \to t'$ be a phase. In this case, we have

$$\gamma_{t:t'} \leq \frac{2 \cdot \text{OPT}(I_t)}{\Delta_{t:t'}} \leq \frac{2 \cdot \text{WEIGHT}(S_t)}{(1-\epsilon) \cdot \epsilon \cdot \text{WEIGHT}(S_t)} = O(1/\epsilon).$$

This already implies the upper bound on the total migration factor. □

3.5 Lower bounds and inapproximability results

We will now complement our positive results by showing how much migration is necessary for some of these problems.

3.5.1 General lower bounds for choosing problems

We will begin for now with the dynamic version of these problems, where objects are added but may also be removed. In this setting, the adversary is quite powerful because removing objects only leaves the option to repack our solution while the range of possibilities has become smaller. This scenario is especially difficult when our current solution becomes inefficient and we might have to change the whole leftover solution. An adversary can make sure that we might need to switch between two or more different solutions making a certain amount of migration necessary. In order to prove our lower bounds of necessary migration, we want to create such a scenario. In order to enforce switching between two solutions, we will use two independent instances, which we will define as *alternating instances*:

Definition 3.5.1. Consider some dynamic online choosing problem Π_{on}, two instances I_1 and I_2, and some desired competitive ratio $\beta < 1$. We call the instances I_1 and I_2 alternating instances when there exists $I'_1 \subset I_1$, such that the following properties hold:

- The solutions $S_1 = I_1$ and $S_2 = I_2$ are feasible and henceforth also $S'_1 = I'_1$ is feasible, but any solution S with $S \cap I_1 \neq \emptyset$ and $S \cap I_2 \neq \emptyset$ is infeasible.
- We have that $\text{PROFIT}(I'_1) < \beta \,\text{PROFIT}(I_2) < \beta^2 \,\text{PROFIT}(I_1)$.

The idea behind these alternating instances is, that any algorithm that hopes to be β-competitive can be forced to alternate between solutions of both instances, when the adversary simply adds all objects from I_1 and I_2 and then continues to remove and add again all items from $I_1 \backslash I'_1$. We can further see that if the necessary migration for the solutions of I'_1 and I_2 is large but the migration potential from $I_1 \backslash I'_1$ is small, that this increases the ratio between necessary migration and migration potential.

Lemma 3.5.1. *Let Π_{on} be a dynamic online choosing problem and consider two alternating instances I_1, I_2 for a desired competitive rate $\beta < 1$ with $I'_1 \subseteq I_2$ fulfilling the requirements of definition 3.5.1. When we additionally have that $\beta \,\text{PROFIT}(I_2) \in \Omega(\gamma)$ for some desired migration factor γ and $\text{PROFIT}(I_1) - \text{PROFIT}(I'_1) \leq c$ for some c, then any β-competitive algorithm requires a migration factor of $\Omega(\frac{\gamma}{2(c+1)})$.*

Proof. Consider some β-competitive algorithm A and the following order of events: First add all items from I_1 and A will generate some approximate solution S_1 for I_1. Now, add also all items from I_2 and note that nothing changes. Since by definition $\text{PROFIT}(I_2) < \beta\,\text{PROFIT}(I_1)$, the algorithm does not need to do anything and will ignore the new items. Remember also that adding any of the new items from I_2 would make the solution infeasible.

We now proceed to remove and add again all items of $I_1 \backslash I_1'$ and repeat this N times for some large $N \in \mathbb{N}$. By removing all these items the previous solution S_1 would consequently be reduced to a solution $S_1' \subseteq I_1'$, and as $\text{PROFIT}(I_1') < \beta\,\text{PROFIT}(I_2)$, our algorithm needs to change to a solution $S_2 \subseteq I_2$. When the items are added again, the algorithm also needs to switch from S_2 to a solution of I_1.

Let us now look at the necessary migration and the migration potential. The total migration potential we received is given through the arrival of all items and the repeated removal and re-adding of items and altogether we have migration potential of $\text{PROFIT}(I_1) + \text{PROFIT}(I_2) + N(\text{PROFIT}(I_1) - \text{PROFIT}(I_1')) = \text{PROFIT}(I_1) + \text{PROFIT}(I_2) + 2Nc$. The necessary migration results from the repacking of the solutions of I_1' and I_2. We have to note however, that the necessary migration for the solutions of I_1' might be small or even 0, when the approximate solution S_1 for I_1 does not use any items of I_1'. We know however that for I_2 we at least exchange a full approximate solution which yields a total necessary migration of at least $N\beta\,\text{PROFIT}(I_2)$. For the total migration factor, we now have that:

$$\frac{N\beta\,\text{PROFIT}(I_2)}{\text{PROFIT}(I_1) + \text{PROFIT}(I_2) + 2Nc} \geq \frac{\beta\,\text{PROFIT}(I_2)}{2(c+1)}$$

when N is chosen large enough. □

We can conclude that one very natural way of proving lower bounds merely requires two instances, call them I_1, I_2, with certain properties. For one the adversary needs to have the possibility of being able to switch between instances I_1 and I_2 with low migration potential. If now for these two instances an algorithm create respective solutions S_1, S_2 in a way that switching between S_1 and S_2 becomes necessary, when the adversary switches between I_1, I_2, and changing solutions requires high migration then the mentioned algorithm will inevitably have a high migration factor.

In the following, we want to show how much repacking is necessary when we want to achieve a robust PTAS or rather a $(1-\epsilon)$-competitive algorithm for some iconic choosing problems. We will start with the SUBSETSUM problem and show that our achieved migration of $O(1/\epsilon)$ is indeed optimal by proving a matching lower bound.

3.5.2 Lower bounds on the migration factor for SUBSETSUM

This section examines lower bounds and inapproximability results in various cases of the SUBSETSUM problem to allow a fine-grained view on the hardness of this online problem. To this end, we split the analysis into four parts, depending on the static vs. the dynamic case and whether the instances are *lax*. Being lax means that the first instance can be non-empty, i. e. that we can present a non-empty instance fully at $t = 1$ without contributing any migration potential. The original setting where we start with an empty instance is the *strict* case.

We will show the following lower bounds on the migration factor of an online algorithm for the SUBSETSUM problem with ratio $1-\epsilon$ with $\epsilon \in (0, \frac{1}{2})$:

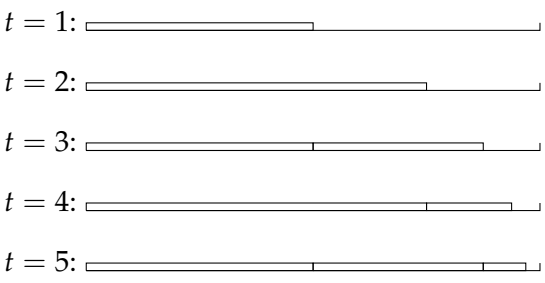
SUBSETSUM	strict	lax
static	$\Omega(\log 1/\epsilon)$	$\Omega(1/\epsilon)$
dynamic	$\Omega(1/\epsilon)$	$\Omega(1/\epsilon)$

Notice that the SUBSETSUM problem is a special case of the vanilla KNAPSACK problem, MULTIPLEKNAPSACK and 2DGEOKNAPSACK. This implies that the lower bounds also hold for these problems. In the preceding sections, we have seen that these problems can be solved using a total migration factor in $O(1/\epsilon)$, such that these bounds are tight except for the strict static case.

Theorem 3.5.2 (Strict static case)**.** *Any online algorithm for the strict static case of* SUBSETSUM *with ratio* $1-\epsilon$ *needs at least a migration factor in* $\Omega(\log 1/\epsilon)$.

Proof. We construct an online instance of the strict static case of SUBSETSUM that generates a migration factor in $\Omega(\log 1/\epsilon)$. To this end, let $T > 0$ be the number of time points in the instance and define the capacity $C := 2^T$. The first instance I_1 contains exactly one item i_1 of size $s_1 = 2^{T-1}$. For later time steps $t-1 \to t$ with $t > 1$, we respectively add items i_t of size $s_t = 3 \cdot 2^{T-t}$.

Consider the following sequence of *alternating solutions*: Whenever t is odd, the solution consists of those items added in odd time steps, i.,e. we have $S_t = \{i_1, i_3, i_5, ..., i_t\}$. When t is even, the solution consists instead of those items added in even time steps, i.e. $S_t = \{i_2, i_4, i_6, ..., i_t\}$. They are visualized in the following diagram:



and so on. In both cases, we see that

$$\text{PROFIT}(S_t) = 2^T - 2^{T-t}.$$

This is the largest number less than $C = 2^T$ that is divisible by 2^{T-t} and hence the best possible profit using items with sizes that are divisible by 2^{T-t}. Note that the total capacity 2^T can never be reached, as all items except i_1 are divisible by three but 2^T is not. This implies that the sequence of alternating solutions described above is optimal and is in fact the only optimal sequence of solutions.

When examining approximate solutions, we observe that every algorithm with ratio $\alpha \in (0,1)$ is forced to choose this sequence of solutions as long as the inequality $\text{OPT}(I_{t-1}) < \alpha \cdot \text{OPT}(I_t)$ holds. Equivalently:

$$\begin{aligned}
\text{OPT}(I_{t-1}) < \alpha \cdot \text{OPT}(I_t) &\iff 2^T - 2^{T-t+1} < \alpha \cdot (2^T - 2^{T-t}) \\
&\iff 1 - 2^{-t+1} < \alpha - \alpha \cdot 2^{-t} \\
&\iff 2^{-t}(\alpha - 2) < \alpha - 1 \\
&\iff t < \log_2 \left(\frac{2-\alpha}{1-\alpha} \right).
\end{aligned}$$

Hence, with $T = \lfloor \log_2((2-\alpha)/(1-\alpha)) - 1 \rfloor$, any algorithm with approximation ratio α will choose this solution. Its migration factor has the following lower bound:

$$\begin{aligned}\gamma &\geq \frac{\sum_{t=1}^{T-1} \text{OPT}(I_t)}{2^{T-1} + \sum_{t=2}^{T} 3 \cdot 2^{T-t}} = \frac{\sum_{t=1}^{T-1}(2^T - 2^{T-t})}{2^{T-1} + \sum_{t=2}^{T} 3 \cdot 2^{T-t}} \\ &= \frac{\sum_{t=1}^{T-1}(1 - 2^{-t})}{2^{-1} + 3 \cdot \sum_{t=2}^{T} 2^{-t}} = \frac{T - 1 - \sum_{t=1}^{T-1} 2^{-t}}{2^{-1} + 3 \cdot \sum_{t=2}^{T} 2^{-t}} \\ &= \Omega(T) = \Omega(\log \tfrac{2-\alpha}{1-\alpha}).\end{aligned}$$

In particular, with $\alpha = 1 - \epsilon$, we get $\gamma = \Omega(\log 1/\epsilon)$. □

Theorem 3.5.3 (Lax static case)**.** *Any online algorithm for the lax static case of* SUBSET-SUM *with ratio* $1 - \epsilon$ *needs at least a migration factor in* $\Omega(1/\epsilon)$*.*

Proof. Let the capacity C be given by $\lfloor 1/\epsilon - 1 \rfloor$. Then

$$\frac{C-1}{C} < 1 - \epsilon.$$

Because we are in the lax case, we are free to present a full instance at $t = 1$ without contributing migration potential. To exploit this, we let the first instance consist of an item i of size $C - 2$ and an item j of size $C - 1$. Because

$$\frac{C-2}{C-1} < \frac{C-1}{C} < 1 - \epsilon,$$

any algorithm with ratio $1 - \epsilon$ will choose the solution consisting of j only. For the next instance, let us insert an item k of size 2. After the insertion, the optimal solution will be $\{i, k\}$ with profit C. The algorithm will follow this sequence of solutions, since

$$\frac{C-1}{C} < 1 - \epsilon.$$

This generates the migration factor

$$\gamma = \frac{C - 2 + C - 1}{2} = \Omega(1/\epsilon).$$

□

We now move on to the strict dynamic case. In the strict case, we are not allowed to present a full instance at $t = 1$ without contributing migration potential. Instead, we have to start with an empty instance and add one item at a time. This destroys the argument from the lax static case. However, we can still salvage the basic parts to construct an instance for the strict dynamic case that generates a migration factor in $\Omega(1/\epsilon)$:

Theorem 3.5.4 (Strict dynamic case)**.** *Any online algorithm for the strict dynamic case of* SUBSETSUM *with ratio* $1 - \epsilon$ *needs at least a migration factor in* $\Omega(1/\epsilon)$*.*

Proof. Let the capacity C be given by $\lfloor 1/\epsilon - 1 \rfloor$. Then, again,

$$\frac{C-1}{C} < 1 - \epsilon.$$

Add an item i of size $C-2$. Clearly, the optimal solution contains exactly this item. Now, we add an item j of size $C-1$. Because

$$\frac{C-2}{C-1} < \frac{C-1}{C} < 1-\epsilon,$$

any algorithm with ratio $1-\epsilon$ will choose $\{j\}$ as the solution. Let us now repeatedly insert and remove an item k of size 2. After each insertion, the optimal solution will be $\{i,k\}$ with profit C. After each deletion, it will be $\{j\}$ with profit $C-1$. The algorithm will follow this sequence of solutions, since

$$\frac{C-1}{C} < 1-\epsilon.$$

Hence, after N repetitions, we moved a load of $NC+N(C-1)$ while only a volume of $C-2+C-1+4N = 2C-3+4N$ was inserted or removed. This generates the migration factor

$$\gamma = \frac{NC+N(C-1)}{2C-3+4N}.$$

For $N \to \infty$, this gives $\gamma \geq (C-1)/4 = \Omega(1/\epsilon)$. □

Corollary. *The same theorem holds for the lax dynamic case of* SUBSETSUM.

3.5.3 Inapproximability of KNAPSACK with weight migration

In the preceding sections, we have assumed that migration is measured by total profit and the migration potential is similarly given by the profit of new items. In this section, we give inapproximability results for the case in which migration is measured by total *weight* and the migration potential is given by the *weight* of new items. Intuitively, we exploit that the KNAPSACK problem is only concerned with maximizing profit, which is a priori uncorrelated with the weight of a solution. It is thus possible to trick an algorithm into migrating a lot of weight for an item with large profit but small weight, i.e. small migration potential.

Theorem 3.5.5 (Lax static case)**.** *There cannot exist an online algorithm with bounded migration factor for the lax static case of* KNAPSACK *with approximation ratio $\alpha > 1/2$ when migration is measured by weight.*

Proof. We construct an online instance of the lax static case that is unsolvable under the hypothesis of a bounded migration factor.

Suppose that there is an online algorithm A_{on} that solves KNAPSACK with an approximation ratio of $\alpha > 1/2$ and assume that the migration factor of A_{on} is bounded by some constant B. Let $C > B$ be the capacity of the knapsack. The instance I_1 at $t=1$ contains the item i with weight C and profit 1. Our algorithm A_{on} will choose the solution $S_1 = \{i\}$. The instance I_2 at $t=2$ is constructed by adding the item j to I_1 with weight 1 and profit 2. Due to $\alpha > 1/2$, this forces the algorithm to choose the solution $S_2 = \{j\}$ and the migration factor

$$\gamma = C > B,$$

is generated. This is in contradiction with the assumption that B be an upper bound on the migration factor. □

Corollary. *The same theorem holds for the following generalizations of the lax static case of* KNAPSACK*:*

- *The lax dynamic case of* KNAPSACK.
- *The lax static and dynamic cases of* MULTIPLEKNAPSACK.
- *The lax static and dynamic cases of* 2DGEOKNAPSACK.

Under the hypothesis that the first instance be empty, the construction in the proof of the previous theorem fails. However, we can reuse the basic ideas for a counterexample in the strict dynamic case and obtain:

Theorem 3.5.6 (Strict dynamic case)**.** *There cannot exist an online algorithm with bounded migration factor for the strict dynamic case of* KNAPSACK *with approximation ratio* $\alpha > 1/2$ *when migration is measured by weight.*

Proof. Let i and j be the same items as in the proof of theorem 3.5.5. We construct an online instance as follows: The first instance is empty as assumed. The second instance I_2 is given by $\{i\}$. In the following time steps, we successively add and remove j to and from the instance for a total of N times. The algorithm has to follow the sequence of optimal solutions by the argument in the proof of theorem 3.5.5. This generates the migration factor

$$\gamma = \frac{NC}{C+N},$$

which is unbounded. □

Corollary. *The same theorem holds for the following generalizations of the strict dynamic case of* KNAPSACK*:*

- *The lax dynamic case of* KNAPSACK.
- *The lax and strict dynamic cases of* MULTIPLEKNAPSACK.
- *The lax and strict dynamic cases of* 2DGEOKNAPSACK.

3.5.4 Lower bounds for maximum independet set

We now take a look at MAXIMUMINDEPENDENTSET again. If we consider MAXIMUMINDEPENDENTSET on arbitrary graphs we can actually prove the same bound for necessary migration, when aiming for a robust PTAS. Since we can choose any selection of edges among nodes in the graph we can simply emulate the same instances that we constructed for the SUBSETSUM Problem.

Theorem 3.5.7. *There is an instance of the online* MAXIMUMINDEPENDENTSET *problem such that the migration needed for a solution with value* $(1-\epsilon)$ OPT *is* $\Omega(1/\epsilon)$*.*

Proof. Set $C := 2\lceil 1/(3\epsilon)\rceil$ and consider two sets of nodes V_1, V_2 with edges $E := \{v_1, v_2 | v_1 \in V_1, v_2 \in V_2\}$, then both V_1 and V_2 are feasible solutions while mixing them would destroy independence. Set $V_1' := V_1 \backslash \{v, w\}$ for two nodes $v, w \in V_1$ and by adding all nodes and then repeatedly removing v and w and re-adding them we created the same situation as for SUBSETSUM leading to a necessary migration factor of $\Omega(1/\epsilon)$. □

While this construction works on arbitrary graphs the same construction can be difficult or rather impossible if we limit ourselves to certain graph classes. If for example we consider MAXIMUMINDEPENDENTSET on planar graphs the same instance cannot be built. Since we have more than three nodes in each of the two sets

V_1, V_2 the graph would have to contain a $K_{3,3}$ as a subgraph and would therefore not be a planar graph. A similar argument is true for unit-disk graphs. When given one node v which we can consider without loss of generality to be in set V_1, then we can see that the number of nodes we can add connected to v but independent to each other is bound by the number of unit-disks we can place non intersecting such that all their center points lie on a disk with radius 2 around the center of v. This number is bound and since we can choose ϵ small enough such that $|V_2|$ exceeds that bound, constructing an instance like above is not possible. This leaves an open problem whether the migration that our algorithm achieved for the cases of MAXIMUMINDEPENDENTSET on planar or unit disk graphs can be improved or whether there exists instances that enforce a migration of $O(1/\epsilon)$.

In the world of weighted MAXIMUMINDEPENDENTSET however it is again possible to prove the lower bound even for graph classes such as unit-disks or planar graphs. We can in this setting once again emulate the same instance that we had for the unweighed case, but can drastically reduce the number of necessary nodes by introducing nodes with high weight, leading to the following result.

Theorem 3.5.8. *There is an instance of the online weighted* MAXIMUMINDEPENDENTSET *problem, containing only three nodes on one path, such that the migration needed for a solution with value* $(1-\epsilon)\,\text{OPT}$ *is* $\Omega(1/\epsilon)$.

Proof. Set $C := 2\lceil 1/(3\epsilon)\rceil$ and consider the set of nodes $V = \{v_1, v_2, v_3\}$ with weights $w_1 := C-2, w_2 := C-1, w_3 := 2$ and $E := \{\{v_1, v_2\}\{v_2, v_3\}\}$. Then the online instance where we add all three nodes v_1, v_2, v_3 and then repeatedly remove and add v_3 is the required instance. □

We can further conclude that this construction works on any graph class that admits a path of length three and therefore solving this problem with a competitive rate of $1-\epsilon$ for $\epsilon < 1$ requires a migration factor of $\Omega(1/\epsilon)$.

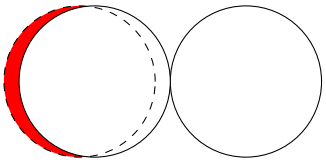
3.5.5 Inapproximability of maximum independet set in unit disk graphs with area migration

We consider the problem of finding maximum independet sets in unweighted unit disk graphs and show that the corresponding online problem cannot be solved with an approximation ratio $\alpha > 1/2$ under the hypothesis of a bounded migration factor *if we assume that the migration potential in the time step* $t-1 \to t$ *is given by the difference in area* between consecutive graphs:

$$\Delta(I_t) = \text{AREA}(I_t) - \text{AREA}(I_{t-1}).$$

Theorem 3.5.9 (Lax static case)**.** *The lax static case of* MAXIMUMINDEPENDENTSET *in unit disk graphs cannot be solved by an algorithm with ratio* $\alpha > 1/2$ *when the migration potential is given by the area difference of consecutive graphs.*

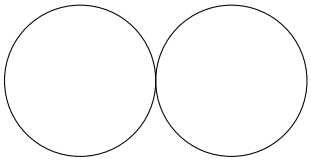
Proof. We construct an online instance that needs an unbounded migration factor in order to be solved. The first graph consists of two unit disks D_1 and D_2 with D_1 tangent to D_2 on the left. Note that the disks overlap. Thus, they can never be part of the same solution. Due to symmetry, we can assume without loss of generality that any algorithm with a ratio $\alpha > 1/2$ will choose the solution $\{D_1\}$. Now, an adversary could add the unit disk D_3 slightly to the left of D_1 in such a way that the area difference (the migration potential) is exactly equal to $\delta > 0$. The situation is summarized in the following picture:



Any approximation algorithm with a ratio $\alpha > 1/2$ will have to choose the solution $\{D_1, D_3\}$, which generates the migration factor $\gamma = 2\pi/\delta$. This is unbounded for $\delta \to 0$. □

Theorem 3.5.10 (Strict dynamic case)**.** *The strict dynamic case of* MAXIMUMINDEPENDENTSET *in unit disk graphs cannot be solved by an algorithm with ratio $\alpha > 1/2$ when the migration potential is given by the area difference of consecutive graphs.*

Proof. In analogy to the lax static case, we construct an instance of the strict dynamic case that generates an unbounded migration factor. The instance is constructed as follows: Add a unit disk D_1. Any approximation algorithm with ratio $\alpha > 1/2$ will choose $\{D_1\}$ as its solution. Then, we add D_2 tangent to D_1 on the right:



To minimize migration, an online algorithm will choose $\{D_1\}$. Again, the adversary could add a unit disk D_3 slightly to the left of D_1 in such a way that the area difference is exactly equal to $\delta > 0$. Now, any algorithm with a ratio $\alpha > 1/2$ has to choose $\{D_2, D_3\}$ as the solution, such that both D_1 and D_2 have to migrate. The adversary then removes D_3 from the instance, generating a migration potential of δ and the algorithm is again forced to choose either D_1 or D_2. To minimize migration, the algorithm would have to choose D_2. In this case, the adversary adds D_4 slightly to the right of D_2 and we would have the same analysis as before. When we play this game for a total of N times, this gives the migration factor

$$\gamma = \frac{2\pi N}{2\pi + 2\delta N}.$$

With $\delta = 1/N$ and $N \to \infty$, this blows up! □

3.6 Framework with complementing Online Algorithm

As we have seen so far, we can solve a range of classical and famous problems in their online versions with small migration by only using an online algorithm. While we have no meaningful result or applications for it, we now want to discuss how to use two algorithms in an alternating fashion and how well the resulting combination performs in terms of migration and competitive rate. The general idea resembles the framework of Berndt *et. al.*. [15] and generalizes the framework we have used so far. Instead of simply waiting between the applications of the offline algorithm in our framework, we now want to bridge the time by applying some known online algorithm. In this way we hope to achieve and use the best results of both worlds: the flexibility of online algorithms and the high solution quality from the offline world. To make this idea work however, we cannot consider any combination of algorithms and we need to ensure that our online algorithm is able to work with the solutions of the offline algorithm. We will call such algorithms *flexible*.

Definition 3.6.1. Let $I \in \Pi_{\text{on}}$ be an instance of the online problem and an online algorithm A_{on} for this problem. Let $t < t' \leq |I|$ be two points of time and S_t be a solution for I_t not necessarily generated by A_{on}. We say an online algorithm is flexible, if it also accepts S_t as a parameter and extends the solution S_t to a solution $S_{t'}$ for $I_{t'}$ by reacting to the events happening in the time interval $t \to \cdots \to t'$. We further say A_{on} has a maintaining ratio of β from t to t', when $A_{\text{off}}(I_t) = \text{PROFIT}(S_t) \geq \alpha \,\text{OPT}(I_t)$ implies that $A_{\text{on}}(I_{t'}, S_t) = \text{PROFIT}(S_{t'}) \geq \alpha\beta \,\text{OPT}(I_{t'})$.

When we combine two algorithms, both delivering approximate solutions, it is inevitable that the solution quality will deteriorate based on both algorithms. We want to try and achieve a final ratio of $\alpha \cdot \beta$ where α is the best known offline approximation ratio and $\beta = 1 - O(\epsilon)$ in order to achieve a similar competitive ratio to the offline result, except a small error of $O(\epsilon)$. We acknowledge therefore that our online algorithm may not uphold this ratio permanently but over some time frame. In this time frame, up to the earliest point of time where the online algorithm would break this desired ratio, we have to exchange our solution for a new better one. On the other side, we also want to achieve a certain migration factor. Therefore, we need to wait long enough until there is a time where the migration costs of exchanging our solution and the migration potential of newly arrived objects or information balances each other out. If such a point of time exists in the time frame where we maintain our desired solution quality, we call this time point a *repacking time*.

Definition 3.6.2. Let $\beta, \gamma \in \mathbb{Q}_{>0}$ and let A_{on} be a flexible online algorithm for Π_{on}. Let t be any point of time, S_t some solution for I_t. Let t' be the first point of time, where A_{on} is not able to keep the maintaining ratio β. If we then have for some $t < t^\star \leq t'$ that for some other β-competitive solution $S'_{t^\star}$ that $\frac{\phi(S_{t^\star-1} \to S'_{t^\star})}{\Delta_{t:t^\star}} \leq \gamma$, we call $t^\star$ a (β, γ)-repacking time and say that A_{on} starting with solution S_t admits a repacking time with maintaining ratio of β and migration γ for $S'_{t^\star}$.

This definition is quite powerful, and we do not require these properties for any arbitrary solutions. For our framework it is important that our chosen online and offline algorithms can work cooperatively. We require that the online algorithm starting with an offline solution maintains the desired competitive ratio until the offline algorithm computes another solution that we can afford migrating to. In that sense, we introduce the term of *compatibility*. If an online algorithm working on a solution of an offline algorithm always admits a (β, γ)-repacking time for some future solution of the offline algorithm, we then call both algorithms *compatible*.

Definition 3.6.3. Let A_{on} and A_{off} be an online and an offline algorithm for Π. We say A_{on} and A_{off} are compatible with maintaining ratio β and migration factor γ if A_{on} starting with some solution S from A_{off} admits a (β, γ)-repacking time for some future solution S' of A_{off}.

We will often mention the time frame from one repacking time until the next and we will regard such a time window as *phase*. During a phase, we will use the online algorithm and handle the changes of the instance until a repacking time occurs. When this happens, we basically want to switch to the solution of the offline algorithm. It may happen that we need to exchange parts or maybe even the complete old solution. The combination of two compatible algorithms achieves a competitive rate dependent on both the approximation ratio of the offline algorithm and the maintaining ratio.

Theorem 3.6.1. *Let A_{off} be an offline algorithm with an approximation ratio of α and A_{on} be an online algorithm compatible with A_{off} maintaining ratio β and migration factor γ.*

Then the combined algorithm F is an online algorithm with a competitive rate of $\alpha \cdot \beta$ and migration factor γ.

Proof. Note that at a (β, γ)-repacking time $t^\star$, we obviously have, due to the approximation ratio of A_{off}, that $F(I_{t^\star}) = A_{\text{off}}(I_{t^\star}) \geq \alpha \operatorname{OPT}(I_{t^\star})$. Similarly, we have for any non-repacking time t, whose last previous repacking time is $t^\star$, that $F(I_t) = A_{\text{on}}(I_t, S_{t^\star}) \geq \alpha\beta \operatorname{OPT}(I_t)$ due to the definition of A_{on}. For the migration factor, we consider each phase from one repacking time $t_1^\star$ to $t_2^\star$ and w.l.o.g. consider $t = 1$ to be the first repacking time as we can simply start with an offline solution. By definition of compatibility with maintaining ratio and migration factor, we know that $\frac{\phi(S_{t_2^\star - 1} \to S_{t_2^\star})}{\Delta_{t_1^\star : t_2^\star}} \leq \gamma$. For the final migration factor, consider any point of time t and say that t lies after k repacking times. Denote with $t_i^\star$ those repacking times and with $S_{t_i^\star - 1}$ the respective repacked solutions and with $S_{t_i^\star}$ the solutions after each repacking for $1 \leq i \leq k$, where S_0 is the empty solution. We then have that our migration factor is bounded by $\frac{\sum_{i=2}^{k} \phi(S_{t_i^\star - 1} \to S_{t_i^\star})}{\Delta_{0:t}} \leq \frac{\sum_{i=2}^{k} \phi(S_{t_i^\star - 1} \to S_{t_i^\star})}{\sum_{i=1}^{k} \Delta_{t_{i-1}^\star : t_i^\star}} \leq \gamma$. Note that we start summing up migration costs at $i = 2$ since at the first repacking time we simply start with an offline solution and hence have no migration costs. □

Overall we end up with a very simple framework. All we need is two algorithms that we apply in an alternating fashion as long as we are able to repack our solutions and balance migration potential and migration costs.

3.7 Conclusion

In this section, we present a general framework to transfer approximation algorithms for many maximization problems to the semi-online setting with bounded migration. Furthermore, we show that the algorithms constructed this way achieve optimal migration. We expect our framework to be also applicable to other problems such as 2DGeoKnapsack variants with more complex objects [75, 53], 3DGeoKnapsack [37], DynamicMapLabeling [17], ThroughputScheduling [33], or MaxEdgeDisjointPaths [45].

With this we have complemented the result of the first section, showing that our framework is applicable for both maximization but also minimization problems.

Chapter 4

Convolution and Knapsack in Higher Dimensions

In the Knapsack problem, one is given the task of packing a knapsack of a given size with items in order to gain a packing with a high profit value. As one of the most classical problems in computer science, research for this problem has gone a long way. One important connection to the $(\max, +)$-convolution problem has been established, where knapsack solutions can be combined by building the convolution of two sequences. This observation has been used in recent years to give conditional lower bounds but also parameterized algorithms.

In this section we want to carry these results into higher dimensions. We consider Knapsack where items are characterized by multiple properties - given through a vector - and a knapsack that has a capacity vector. The packing must now not exceed any of the given capacity constraints. In order to show a similar sub-quadratic lower bound we introduce a multi-dimensional version of Convolution as well. We will generalize this problem and combine higher dimensional matrices. We then consider variants of this problem introduced by Cygan et al. and prove that they are all equivalent in terms of algorithms that allow for a running time sub-quadratic in the number of entries of the matrix.

We further develop a parameterized algorithm to solve higher dimensional Knapsack. The techniques we apply are inspired by an algorithm introduced by Axiotis and Tzamos. In general, we manage not only to extend their result to higher dimensions, but also simplify their algorithm by removing some step. We will show that even for higher dimensional Knapsack, we can reduce the problem to convolution on one-dimensional sequences, leading to an $\mathcal{O}(d(n + D \cdot \max\{\Pi_{i=1}^{d} t_i, t_{\max} \log t_{\max}\}))$ algorithm, where D is the number of different weight vectors, t the capacity vector and d is the dimension of the problem. Finally we also modify this algorithm to handle items with negative weights to cross the bridge from solving not only Knapsack but also Integer Linear Programs (ILPs) in general.

4.1 Introduction

The Knapsack problem is one of the core problems in computer science. The task of finding a collection of items that fits into a knapsack but also maximizes some profit has proven to be quite difficult. In fact, this problem is NP-hard and as such the aim is to find approximation algorithms and determine lower bounds for the running time of exact algorithms. Despite being a very old problem, new results are being made even to this day.

Cygan et al. [36] and Künnemann et al [71] among others have used the following relationship between Knapsack and the $(\max, +)$-convolution problem. Assume

we are given two disjoint item sets A and B and a knapsack of size t. If we additionally know the optimal profits for all knapsack sizes $t' \leq t$ we then can calculate the maximum profits for all sizes t' for $A \cup B$ by using convolution. This connection was used in order to show that the existence of a sub-quadratic algorithm for Knapsack (sub-quadratic in the number of possible knapsack capacities) is equivalent to the existence of a sub-quadratic algorithm for $(\max, +)$-convolution.

In this work, we consider these problems in higher dimensions. The Knapsack problem is simply generalised by replacing the single value weight and capacity by vectors. We then look for a collection of items whose summed up vectors do not exceed the capacity vector in any position. The natural question arises whether a similar quadratic time lower bound exists for this problem. We answer this question positively by giving a generalisation of convolution in higher dimensions as well. Using this generalisation and techniques introduced by Bringmann [21] such as Color-Coding we are able to achieve similar subquadratic lower bounds as in the one-dimensional case.

Additionally, we also give a new parameterized algorithm that solves multidimensional Knapsack in linear time in the number of different item weights and the number of all feasible knapsack capacities. This algorithm is mainly a generalisation of an algorithm introduced by Axiotis and Tzamos [2]. We show how, in a similar style to their algorithm, one can combine one-dimensional sequences to a solution of the higher dimensional problem. This allows to build higher dimensional convolutions for this special case in linear time.

Lastly we will also apply these results to knapsack with negative item weights and later to integer linear programs as well. Using the concept of proximity with the bound from Eisenbrand and Weismantel [41] we also give an algorithm for ILPs.

4.1.1 Problem Definitions and Notations

We denote with $[k] := \{i \in \mathbb{N} \mid 1 \leq i \leq k\}$. In the following, we write for two vectors $v, u \in \mathbb{R}^d$ that $v \leq u$ (resp. $v < u$) if for all $i \in [d]$ we have that $v_i \leq u_i$ (resp. $v_i < u_i$). Further we denote with $v_{max} = \max_{i \in [d]} v_i$ for any vector $v \in \mathbb{R}^d$. We define with $\overrightarrow{\mathbf{k}} \in \mathbb{R}^d$ the vector that has $k \in \mathbb{R}$ in every position. We therefore can write the zero vector as $\overrightarrow{\mathbf{0}}$.

Definition 4.1.1. Let $L \in \mathbb{N}^d$ be a d-dimensional vector and $A = (A_{i_1 i_2 \cdots i_d})$ be an $L_1 \times L_2 \times \cdots \times L_d$ matrix. We call the vector L the size of A and denote the number of entries in A with $\Pi(L) := \prod_{i=1}^{d} L_i$.

We call a vector $v \in \mathbb{Z}$ with $\overrightarrow{\mathbf{0}} \leq v \leq L - \overrightarrow{\mathbf{1}}$ *position* of matrix A. For ease of notation, we will denote for any matrix position v of A the respective matrix entry $A_{v_1 v_2 \cdots v_d}$ with A_v.

We note that in this definition, matrix positions lie in between $\overrightarrow{\mathbf{0}}$ and $L - \overrightarrow{\mathbf{1}}$. This makes it easier to work with positions for convolution and also for formulating time complexity bounds, as $\Pi(L)$ will be the main parameter we consider.

The core problem we will work with is the maximum convolution problem. In the one dimensional case of this problem, one is given two sequences a, b of length n and has to calculate a third sequence c of the same length. The k-th entry of c is supposed to be the maximum combination of two values from a and b whose indices add up to k. Precisely $c_k := \max_{i \in [k]} a_i + b_{k-i}$. For the higher dimensional generalisation, we consider a d-dimensional matrix as per Definition 4.1.1. We define maximum convolution in the same spirit as the one-dimensional case for these matrices.

Definition 4.1.2 (Maximum Convolution)**.** Given two d-dimensional matrices A, B with equal size L, the $(\max, +)$-convolution of A and B denoted as $A \oplus B$ is defined as a matrix C of size L with with $C_v := \max_{u \leq v} A_u + B_{v-u}$ for any $v < L$.

d-MAXCONV	
Input:	Two d-dimensional matrices A, B with equal size L.
Problem:	Compute the matrix $C := A \oplus B$ of size L.

Note that in the following we will refer to this problem as "Convolution". We specifically limit ourselves to the special case where both input matrices and the output matrix have the same size. In a more general setting, we could allow matrices of sizes $L^{(1)}, L^{(2)}$ as input and compute a matrix of size $L^{(1)} + L^{(2)} - \overrightarrow{\mathbf{1}}$ (remark that the subtraction is required due to the way we defined sizes).

One can however reduce this general case to the case of same sizes by simply extending each matrix to the same size. The common size of every matrix in each dimension $i \in [d]$ is then $\max(L_i^{(1)}, L_i^{(2)})$. The new positions are simply filled with a small enough value $-\infty$. Any position in the resulting matrix that holds $-\infty$ then has no combination from the original input matrix entries.

To measure the running time of our algorithms, we mainly consider the size or rather the number of entries from the resulting matrix - because we need to calculate a value for every position. Therefore, in terms of theoretical performance, working with different sizes or calculating a matrix of combined size will not make a difference. By only considering matrices of the same size, we avoid many unnecessary cases and the dummy values of $-\infty$.

We note that all of our reductions can be applied even for matrices with different sizes. It will mostly be sufficient to use the reduction mentioned above. If further steps are required, we will mention them when discussing the specific reduction.

We can conclude for d-MAXCONV that there is a quadratic time algorithm like in the 1-dimensional case. This algorithm simply iterates through all positions that need to be calculated and tries out all combinations in time $\mathcal{O}(\Pi(L)^2) \subseteq \mathcal{O}(L_{max}^{2d})$.

Further we define an upper bound test for the convolution problem. In this problem, we are given a third input matrix and need to decide whether its entries are upper bounds for the entries of the convolution.

d-MAXCONV UPPERBOUND	
Input:	Three d-dimensional matrices A, B, C with equal size L.
Problem:	Decide whether $(A \oplus B)_v \leq C_v$ for all $v < L$.

We further generalise the notion of superadditivity to these matrices.

Definition 4.1.3. Given one d-dimensional matrix A of size L, we call A superadditive when for all positions $v < L$ we have that $A_v \geq (A \oplus A)_v$.

d-SUPERADDITIVITY TESTING	
Input:	One d-dimensional matrix A of size L.
Problem:	Decide whether A is superadditive.

The next problem class we consider is the d-KNAPSACK problem. In the one dimensional case, one is given a set of items with weights and profits and one knapsack with a certain weight capacity. The goal is now to pack the knapsack such that

the profit is maximized and the total weight of packed items does not exceed the weight capacity.

The natural higher dimensional generalisation arises when we have more constraints to fulfill. When going on a journey by plane, one for example has several further requirements such as a maximum amount of allowed liquid or number of suitcases. By imposing more similar requirements, we can simply identify each item by a vector and also define the knapsack by a capacity vector. This leads to the following generalisation of Knapsack into higher dimensions. We further differentiate between two problems 0/1 d-KNAPSACK and UNBOUNDED d-KNAPSACK, depending on whether we allow items to be only used one time or an arbitrary number of times.

0/1 d-KNAPSACK	
Input:	Set I of n items each defined by a profit $p_i \in \mathbb{R}_{\geq 0}$ and weight vector $w^{(i)} \in \mathbb{N}^d$, along with a knapsack of capacity $t \in \mathbb{N}^d$.
Problem:	Find subset $S \subseteq I$ such that $\sum_{i \in S} w^{(i)} \leq t$ and $\sum_{i \in S} p_i$ is maximal.

UNBOUNDED d-KNAPSACK	
Input:	Set I of n items each defined by a profit $p_i \in \mathbb{R}_{\geq 0}$ and weight vector $w^{(i)} \in \mathbb{N}^d$, along with a knapsack of capacity $t \in \mathbb{N}^d$.
Problem:	Find a multi-set $S \subseteq I$ such that $\sum_{i \in S} w^{(i)} \leq t$ and $\sum_{i \in S} p_i$ is maximal.

The running time for these problems is mainly dependent on the dimension d, the number of items n of an instance and the number of feasible capacities $\Pi(t + \overrightarrow{\mathbf{1}})$ and we will further study the connection of these in regards to the convolution problems.

4.1.2 Related Work

Cygan et al. [36] as well as Künnemann [71] initiated the research and were the first to introduce this class of problems and convolution hardness. In particular, Cygan et. al. showed for the case of $d = 1$ that all these problems are equivalent in terms of whether they allow for a subquadratic algorithm. This was mainly done to formulate a conditional lower bound for all these problems under the hypothesis that no subquadratic algorithm for d-MAXCONV exists.

Conjecture 4.1.1 (MaxConv-hypothesis)**.** There exists no $\mathcal{O}(\Pi(L)^{2-\epsilon})$ time algorithm for any $\epsilon > 0$ for d-MAXCONV with $d = 1$.

The best known algorithm to solve convolution on sequences without any further assumption takes time $n^2/2^{\Omega(\sqrt{\log n})}$. This result was achieved by Williams [82], who gave an algorithm for APSP in conjunction with a reduction by Bremner et al. [20]. However, the existence of a truly subquadratic algorithm is yet an open question.

Research therefore has taken a focus on special cases of convolution where one or both input sequences is required to have certain structural properties such as monotonity, concavity or linearity. Chan and Lewenstein [28] gave a subquadratic $\mathcal{O}(n^{1.864})$ algorithm for instances where both sequences are monotone increasing and the values are bound by $\mathcal{O}(n)$. Chi et al. [31] improved this further with a randomized $\widetilde{\mathcal{O}}(n^{1.5})$ algorithm.

Axiotis and Tzamos as part of their Knapsack algorithm showed that Convolution with only one concave sequence can be solved in linear time $\mathcal{O}(n)$ [2]. Gribanov et al. [54] studied multiple cases and gave subquadratic algorithms when one sequence is convex, piece-wise linear or polynomial.

Furthermore, Convolution has proven to be a useful tool to solve other problems as well, in particular the Knapsack problem. In fact one of the reductions of Cygan et al. [36] was an adapted version of Bringmann's algorithm for subset sum [21]. Bringmann's algorithm works by constructing sub-instances, solving these and then combining the solutions via Fast Fourier Transformation (FFT). The algorithm by Cygan et al. to solve Knapsack works the same way, but uses Convolution instead of FFT.

Axiotis and Tzamos follow a similar approach but choose their sub-instances more carefully, by grouping items with the same weight. That way, the solutions make up concave profit sequences which can be combined in linear time [2]. This yields in total an $\mathcal{O}(n + Dt)$ algorithm, where D is the number of different item weights.

Polak et al. [77] gave an $\mathcal{O}(n + \min\{w_{max}, p_{max}\}^3)$ algorithm, where w_{max}, p_{max} denote the maximum weight and profit respectively. They achieved this by combining the techniques from Axiotis and Tzamos with proximity results from Eisenbrand and Weismantel [41]. Further, Chen et al. [30] improved this to a time of $\widetilde{\mathcal{O}}(n + w_{max}^{2.4})$. Very recently, Ce Jin [66] gave an improved $\mathcal{O}(n + w_{max}^2 \cdot \log^4 w_{max})$ algorithm. Independently to these results for $0-1$-Knapsack, Bringmann gave an $\widetilde{\mathcal{O}}(n + w_{max}^2)$ algorithm for Bounded Knapsack where items may be taken multiple times, but not arbitrarily often.

For the multi-dimensional setting, we would like to note that Knapsack problems are a special case of integer linear programs.

INTEGER LINEAR PROGRAM (ILP)	
Input:	A matrix $A \in \mathbb{Z}^{d \times n}$, a target vector $b \in \mathbb{Z}^d$, a profit vector $c \in \mathbb{Z}^n$ and an upper bound vector $u \in \mathbb{N}^d$.
Problem:	Find $x \in \mathbb{Z}^n$ with $Ax = b, 0 \leq x \leq u$ and such that $c^T x$ is maximal.

If we limit A to be a matrix with only non-negative entries and $u := \overrightarrow{\mathbf{1}}$ then solving the above defined ILP is equivalent to $0/1$ d-KNAPSACK. If we omit the $x \leq u$ constraint, the resulting problem is UNBOUNDED d-KNAPSACK.

A very important part in research of ILPs has been on the so-called *proximity*. The proximity of an ILP is the distance between an optimal solution and an optimal solution of its relaxation. The relaxation of an ILP is given by allowing the solution vector x to be non-integral, that is $x \in \mathbb{R}^n$.

Eisenbrand and Weismantel [41] have proven that for an optimal solution of the LP-relaxation $x^* \in \mathbb{R}^n$ there is an optimal integral solution $z^* \in \mathbb{Z}^n$ with $||x^* - z^*||_1 \leq d(2d\Delta + 1)^d$ with Δ being the largest absolute entry in A. They used this result to give an algorithm that solves ILPs without upper bounds in time $\mathcal{O}(n \cdot \mathcal{O}(d\Delta)^{dm} \cdot \|b\|_1^2)$ and ILPs as defined above in time $\mathcal{O}(n \cdot \mathcal{O}(d)^{(d+1)^2} \cdot \mathcal{O}(\Delta)^{d(d+1)} \cdot \log^2(d\Delta))$. They additionally extended their result to Knapsack in particular and optimized the running time to $\mathcal{O}(n^2 \cdot \Delta^2)$ for Knapsack where item i may be taken up to u_i times.

There have been further results on proximity such as Lee et al. [72], who gave a proximity bound of $3d^2 \log(2\sqrt{d} \cdot \Delta_m^{1/m}) \cdot \Delta_m$ where Δ_m is the largest absolute value of a minor of order m of matrix A. Celaya et al. [26] also gave further proximity bounds for certain modular matrices.

4.1.3 Our Results

We begin by expanding the results of Cygan et al. [36] into higher dimensions. The natural question is whether similar relations shown in their work also exist in higher dimensions and in fact they do. In the first part of this section we will show that the same equivalence - in regards to existing subquadratic time algorithms - holds among the higher dimensional problems, at least for fixed dimension d.

Theorem 4.1.1. *For some fixed dimension d and any $\epsilon > 0$, the following statements are equivalent:*

1. *There exists an $\mathcal{O}(\Pi(L)^{2-\epsilon})$-time algorithm for d-MaxConv.*
2. *There exists an $\mathcal{O}(\Pi(L)^{2-\epsilon})$-time algorithm for d-MaxConv UpperBound.*
3. *There exists an $\mathcal{O}(\Pi(L)^{2-\epsilon})$-time algorithm for d-SuperAdditivity Testing.*
4. *There exists an $\mathcal{O}(n + \Pi(t)^{2-\epsilon})$-time algorithm for Unbounded d-Knapsack.*
5. *There exists an $\mathcal{O}(n + \Pi(t)^{2-\epsilon})$-time algorithm for $0/1$ d-Knapsack.*

Some of these reduction incur a multiplicative factor of $\mathcal{O}(2^d)$. This is a natural consequence due to the exponentially larger amount of entries that our matrices hold and that we need to process. As an example, where it was sufficient in the 1-dimensional case to split a problem in two sub-problems, we may now need to consider 2^d sub-problems. For this reason we require d to be fixed so we can omit these factors. We will prove this statement through a ring of reductions. We note that one of the reductions uses an algorithm or a generalisation of it from Bringmann [21, 2] that is randomized. Part of this algorithm, involving so-called Color-Coding can be derandomized, but a full derandomization is still an open problem.

We note that under the MaxConv-hypothesis, there does not exist an $\mathcal{O}(\Pi(L)^{2-\epsilon})$-time algorithm for 1-MaxConv. If there exists any d such that d-MaxConv admits a sub-quadratic algorithm, then it would also solve the problem in sub-quadratic time for any $d' \leq d$, hence contradicting the MaxConv-hypothesis, because we can extend any d'-dimensional matrix to a d-dimensional one.

We complement our conditional lower bound with a parameterized algorithm. To achieve this, we generalize an algorithm by Axiotis and Tzamos [2]. Our algorithm will also have a running time dependent on the number of different item weights, that we denote with D and the largest item weight $\Delta := \max_{i \in I} ||w^{(i)}||_\infty$. This algorithm will also group items by weight vector and solve the respective sub-instances. The resulting partial solution are then combined via d-MaxConv.

As per our first result however, solving d-MaxConv needs quadratic time in the number of entries. We can overcome this barrier however, by reducing our problem to convolution on 1-dimensional sequences. To see how solutions from specifically generated sequences can make up the complete solution for d-MaxConv is more intricate then in the 1-dimensional case. We consider therefore a slightly different approach then Axiotis and Tzamos, by solving a knapsack variant where solutions need to have exactly the demanded weight. This even allows us to simplify our algorithm and omit a sliding window technique that Axiotis and Tzamos used in their algorithm, making our algorithm a bit more light weight.

Theorem 4.1.2. *There is an algorithm for $0/1$ d-Knapsack with running time* $\mathcal{O}(d(n + D \cdot \max\{\Pi(t), t_{\max} \log t_{\max}\})) \subseteq \mathcal{O}(d(n + (\Delta+1)^d \cdot \max\{\Pi(t), t_{\max} \log t_{\max}\}))$.

In the general case our algorithm will achieve a running time of $\mathcal{O}(d(n + D \cdot \Pi(t))$ as the $t_{\max} \log t_{\max}$ part only becomes relevant when the capacity t indicates a slim knapsack structure, that is very large in one component, but comparatively small in the other. We argue that the slim case, while potentially allowing for more efficient algorithms, is more unlikely because with growing number of constraints d we are more likely to have two large capacities. We note that our algorithm achieves the lower bound proposed in Theorem 4.1.1 since D is also upper bounded by $\Pi(t)$.

We further extend this algorithm to work on ILPs. Our algorithm achieves a similar running time to Eisenbrand and Weismantel with a time of $n + 2D \cdot \mathcal{O}(\Delta)^{(d+1)^2} \cdot \mathcal{O}(d)^{d(d+2)}$. That is slightly worse in Δ, slightly better in d. Most importantly however our algorithm is mostly independent on n but relies on the number of different columns of the given ILP.

4.2 Reductions

For the Convolution problems, we will formulate the running time via $T(\Pi(L), d)$, where d is the dimension and L is the size of the result matrix - meaning the first parameter resembles the number of entries in the matrix. For the Knapsack problems, we will add the number of items n as parameter and denote the running time of an algorithm via $T(n, \Pi(t + \overrightarrow{\mathbf{1}}), d)$, again using $\Pi(t + \overrightarrow{\mathbf{1}})$ to denote the number of possible knapsack capacities.

Similar to Cygan et al. [36], we mainly look at functions satisfying $T(\Pi(L), d) = c^{\mathcal{O}(d)} \cdot \Pi(L)^{\alpha}$ for some constants $c, \alpha > 0$. Therefore, we remark that for all constant $c' > 1$ we can write $T(\Pi(c' \cdot L), d) \in \mathcal{O}(T(\Pi(L), d))$ since d is fixed and we then have $\Pi(c'L)^{\alpha} = c'^{d \cdot \alpha} \cdot \Pi(L)^{\alpha}$.

For all the mentioned problems we assume that all inputs, that is also any value in any vector, consist of integers in the range of $[-W, W]$ for some $W \in \mathbb{Z}$. This W is generally omitted as a running time parameter and polylog(W) are omitted or hidden in T functions. We remark that - unavoidably - during the reductions we may have to deal with larger values than W. We generally will multiply a factor of polylog(λ) in cases where the values we have to handle increase up to λW. Note that if λ is a constant, we again omit it.

We follow the same order as Cygan et al. [36], because that makes the reduction increasingly more complex. For the first reduction from UNBOUNDED d-KNAPSACK to $0/1$ d-KNAPSACK, we use the same reduction as in the one-dimensional case. The idea is to simply encode taking a multitude of items via binary encoding.

Theorem 4.2.1 (UNBOUNDED d-KNAPSACK $\rightarrow$ $0/1$ d-KNAPSACK)**.** *A $T(n, \Pi(t), d)$ algorithm for $0/1$ d-KNAPSACK implies an $\mathcal{O}(T(n \log(t_{max}), \Pi(t), d))$ algorithm for* UNBOUNDED d-KNAPSACK, *where $t \in \mathbb{N}^d$ is the capacity of the knapsack.*

Proof. We denote the i-th item by the pair $(w^{(i)}, p_i)$ where $w^{(i)} \in \mathbb{N}^d$ is the weight vector and $p_i \in \mathbb{R}_{\geq 0}$ is the profit. Consider an instance for the unbounded Knapsack problem and construct a 0/1-Knapsack instance as follows. For each item $(w^{(i)}, p_i)$ add items $(2^j w^{(i)}, 2^j p_i)$ for all j such that $2^j w^{(i)} \leq t$.

Without loss of generality, we may assume that all n weight vectors are pairwise different. If two weight vectors are the same, we would take the item with the higher profit. For each of these items in the unbounded instance, we then have at most $\lfloor \log(t_{max}) \rfloor + 1$ items in the constructed bounded instance.

Assume now that in the unbounded instance some item i is taken k times in some solution. This choice can be replicated by taking the copies from i that make up the

binary representation of k. With this, we can convert any packing of either instance to a packing of the other. Therefore, both instances are equivalent. □

For the next reduction we reduce d-SUPERADDITIVITY TESTING to a special case where each matrix entry is non-negative and values fulfill a monotony property.

Definition 4.2.1. For a d-dimensional matrix A with size $L \in \mathbb{N}^d$, we say that A is monotone increasing if for any positions $\overrightarrow{\mathbf{0}} \leq v, u < L$ we have that $v \leq u$ implies that $A_v \leq A_u$.

When we test for superadditivity, which is merely testing whether a matrix A is an upperbound for the convolution of $A \oplus A$, it is helpful if we can work with this simpler structure.

Lemma 4.2.2. *For every d-dimensional matrix A of size $L \in \mathbb{N}^d$, there is a matrix A' with the same size that is non-negative and monotone increasing such that further A is superadditive iff A' is superadditive. The values in this new matrix may increase by a factor of $\mathcal{O}(dL_{max})$.*

Proof. Initially, set the entry at the zero vector position as $A'_{\overrightarrow{\mathbf{0}}} := \max\{0, A_{\overrightarrow{\mathbf{0}}}\}$. Note that if $A_{\overrightarrow{\mathbf{0}}} > 0$ then A cannot be superadditive since already $A_{\overrightarrow{\mathbf{0}}} + A_{\overrightarrow{\mathbf{0}}} > A_{\overrightarrow{\mathbf{0}}}$. Set $c := 2\max_{v \leq L} |A_v| + 1$ and with that define $A'_v := A_v + c \cdot ||v||_1$ for all $v < L$.

We can immediately conclude that through the setting of c we have that $A_v + c > 0$, hence A' has no negative values. The monotony follows from the fact that $|A_v - A_{v+e^{(i)}}| < c$ for any unit vector $e^{(i)}$ and with that we have

$$A'_v - A'_{v+e^{(i)}} = A_v + \|v\|_1 c - (A_{v+e^{(i)}} + \left\|v + e^{(i)}\right\|_1 c) = A_v - A_{v+e^{(i)}} - c < c - c = 0.$$

We now show the equivalence of superadditivity between the two matrices for any positions $u, v \neq \overrightarrow{\mathbf{0}}$ such that $u + v$ is a feasible position in the matrix A. Remember that the positions are non-negative, so $\|u + v\|_1 = \|u\|_1 + \|v\|_1$ and therefore:

$$\begin{aligned} & A'_v + A'_u \leq A'_{v+u} \\ \Leftrightarrow\ & A_v + c \cdot ||v||_1 + A_u + c \cdot ||u||_1 \leq A_{v+u} + c \cdot ||v+u||_1 \\ \Leftrightarrow\ & A_v + A_u \leq A_{v+u} \end{aligned}$$

Consider now the case that one vector is actually the null-vector. Without loss of generality assume that $u = \overrightarrow{\mathbf{0}}$ and let v be arbitrary, but such that $u + v$ is still a feasible position. We then have that $A'_u = 0$ if and only if $A_u \leq 0$. The equality follows then immediately:

$$\begin{aligned} & A'_v + A'_u = A'_{v+u} = A'_v \\ \Leftrightarrow\ & A_v + A_u \leq A_{v+u} = A_v \end{aligned}$$

Let L be the size of A and remember that our input values through our initial assumption are all in $[-W, W]$. Let A'_p with $p = L - \overrightarrow{\mathbf{1}}$ be the largest entry in A'. We note that the values of our new matrix may grow up to

$$A'_p = A_p + c \cdot \|p\|_1 \leq W + (2W + 1) \cdot \left\|L - \overrightarrow{\mathbf{1}}\right\|_1 \in \mathcal{O}(||L||_1 W) \in \mathcal{O}(dL_{max}W).$$

□

Both properties, monotony and non-negative values, play an important role in the reduction to UNBOUNDED d-KNAPSACK, where we translate matrix positions to knapsack items.

Theorem 4.2.3 (d-SUPERADDITIVITY TESTING $\rightarrow$ UNBOUNDED d-KNAPSACK)**.** *A $T(n, \Pi(t), d)$ algorithm for* UNBOUNDED d-KNAPSACK *implies an algorithm that tests d-*SUPERADDITIVITY TESTING *for a matrix A with size L in time $\mathcal{O}(T(2\Pi(L), \Pi(2L), d)\,\text{polylog}\,(dL_{max}))$.*

Proof. Let A be an input matrix with size L and assume thanks to Theorem 4.2.2 that A is non-negative and monotone increasing. This might cause larger entries in the matrix that might incur additional overhead. We will account for this by adding a factor $\text{polylog}(dL_{max})$ to the overall running time.

Let $D := ||L||_1 \cdot \max_{v<L} A_v + 1$. Set $t := 2L$ as the new capacity for our knapsack instance. For every position $v < L$ in A, we define two items for our Knapsack instance. The first item, the primal item of position v, will have weight vector $w^{(v)} = v$ and profit $p_v = A_v$. The other item, the dual item of position v will have weight vector $\overline{w}^{(v)} = t - v$ and profit $\overline{p}_v = D - A_v$.

We can see that the profit D is easily achievable for this Knapsack instance as any combination of primal and dual item for some position will yield a feasible packing that fills out the whole knapsack. We further argue that D is the maximum achievable profit if and only if A is a superadditive matrix.

First consider the case where A is not superadditive, which means for some $u, v < L$ we have that $A_v + A_u > A_{u+v}$. Combine now the primal items for positions u, v with the dual item for position $u + v$. By definition of the weights the total weight of this packing is t. The profit of this packing then sums up to: $D - A_{v+u} + A_v + A_u > D$.

Consider now the case of A being superadditive. We first argue that any knapsack solution may hold at most one dual item. Consider therefore any $1 \leq i \leq d$ and note that $(t - v)_i > 2L_i - L_i = L_i$ for any position v. Therefore, adding two dual items would break all capacity constraints. Further, any optimal solution has to contain a dual item.

We may assume that no item has weight $\overrightarrow{\mathbf{0}}$, because these items can be removed and automatically added to the knapsack. This also means that there can be at most $||L||_1$ primal items in the knapsack and that the maximum profit of a packing containing only primal items is limited to $||L||_1 \cdot \max_{v<L} A_v < D$.

Let S now be an optimal solution containing the dual item for position v. Assume further that S contains multiple primal items and take two of these primal items for positions u and w. Due to the superadditivity of A, we have that $A_u + A_w \leq A_{u+w}$, so we can replace both of these primal items with the primal item of position $u + w$. This will not change the weight of the knapsack and only increase its profit, so the solution stays optimal. By doing this replacement over and over, we end up with an optimal packing that contains one dual item for position v and only one primal item for some position x.

Note that we must have that $x \leq v$ because otherwise the weight constraint would be broken and since we never changed the weight of the knapsack, this would have been true for the supposedly optimal solution S as well. Since A is monotone, we have that $A_x \leq A_v$, so we can yet again replace the primal item for position x with the primal item of position v and end up with A packing of profit D, which has to be optimal. □

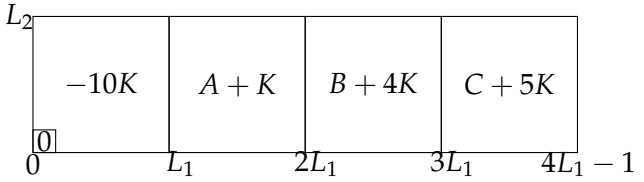


FIGURE 4.1: Structure of the constructed matrix for $d = 2$. Note that the annotation $X + s$ for a matrix X and some value s, denotes the matrix where $(X + s)_v = X_v + s$ for every position v.

The next reduction differs from Cygan et al. [36]. We also combine our input matrices together, but in the context of matrices we need to handle a number of different combinations more than in the 1-dimensional case. We therefore add a block of negative values in an initial dummy block. This way, any combination that is not relevant to the actual upper bound test, will result positively when tested in the upper bound test.

Theorem 4.2.4 (d-MAXCONV UPPERBOUND $\rightarrow$ d-SUPERADDITIVITY TESTING)**.** *A $T(\Pi(L), d)$ algorithm for d-SUPERADDITIVITY TESTING implies an $\mathcal{O}(T(4\Pi(L), d))$ algorithm for d-MAXCONV UPPERBOUND.*

Proof. Let A, B, C be the input matrices with the same size L. Let L_1 be the first entry of L. We will construct a new matrix M of size L' with $L'_1 := L_1 \cdot 4$ and $L'_i = L_i$ for $1 < i \leq d$, where we append each matrix in the first dimension. This means the previous three matrices appear next to each other preceded by a default block with negative values and 0 in the $\overrightarrow{\mathbf{0}}$ position. We formally define this matrix M blockwise and address the four blocks of M via index sets. For $0 \leq i \leq 3$, denote with $P_i := e^{(i)} \cdot iL_1$ the point where each matrix starts in the compound matrix (i.e. this will refer to the $\overrightarrow{\mathbf{0}}$ position in the original matrix) and denote the index set of each matrix as $I_i := \{v < L' | P_i \leq v < P_i + L\}$.

Let K be large enough i.e. $K = 1 + 2\max_{v<L}\{|A_v|, |B_v|, |C_v|\}$, then define for $v < L'$:

$$M_v = \begin{cases} 0 & \text{if } v \in I_0 \text{ and } v = \overrightarrow{\mathbf{0}} \\ -10K & \text{if } v \in I_0 \text{ and } v \neq \overrightarrow{\mathbf{0}} \\ K + A_u & \text{if } v \in I_1 \text{ and } v = 1 \cdot (L_1 + 1) + u \\ 4K + B_u & \text{if } v \in I_2 \text{ and } v = 2 \cdot (L_1 + 1) + u \\ 5K + B_u & \text{if } v \in I_3 \text{ and } v = 3 \cdot (L_1 + 1) + u \end{cases}$$

We now argue that M being superadditive is equivalent to C being an upper bound for the convolution of A and B. To see that, we make a case distinction over all $u, v < L'$ such that $u + v < L'$ and compare the respective entries in M. Note that we will skip symmetrical cases that appear by swapping the roles of v and u.

Case 1: $v = \overrightarrow{\mathbf{0}}$. We now have that $M_v + M_u = M_u = M_{u+v}$, hence the superadditivity property is always fulfilled.

Case 2: $v, u \in I_0 \setminus \overrightarrow{\mathbf{0}}$. In this case we get that $M_v + M_u = -20K < -10K \leq M_{u+v}$ for all u.

Case 3: $v \in I_0 \setminus \overrightarrow{\mathbf{0}}$ and $u \in I_i$ for $i \geq 1$. It follows now $M_v + M_u < 0 < M_{u+v}$ since $u + v \in I_i$ for $i \geq 1$.

Case 4: $v, u \in I_1$. Since two entries from A added up will be smaller than K, we have that $M_v + M_u < 3K \leq M_{u+v}$ since $u + v \in I_i$ for $i \geq 2$.

Case 5: $v \in I_1$ and $u \in I_2$. Note that $u + v \in I_3$ and set v', u', w' such that $M_v = K + A_{v'}$, $M_u = 4K + B_{u'}$ and $M_{u+v} = 5K + C_{w'}$. Through definition we additionally

have that $v' + u' = w'$. We conclude now that $M_v + M_u = K + A_{v'} + 4K + B_{u'} = 5K + A_{v'} + B_{u'}$ and with that we have $M_v + M_u \leq M_{u+v} \Leftrightarrow A_{v'} + B_{u'} \leq C_{w'}$ and hence in this case the superadditivity property is fulfilled iff C is an upper bound for the convolution of A and B.

□

For this reduction, we briefly want to discuss how to handle differently sized matrices, because the construction of M in this case is not clear. Consider matrices A, B, C with sizes $L^{(1)}, L^{(2)}, L^{(3)}$. We assume that C is the largest matrix in any dimension, so for any $1 \leq i \leq d$ we have $L_i^{(3)} \geq \max(L_i^{(1)}, L_i^{(2)})$. If that is not the case it actually opens up the question of what should happen in regards to the upper bound check. Should it fail, because the positions do not exist or should we just reduce the test to the actual matrix that we have? We can solve both these problems, just by filling the matrix C with either indefinitely small or large entries and enforce either result.

When C is the largest matrix we can modify A and B to bring them all to equal size. We simply expand A and B to a matrix of size $L^{(3)}$ and fill up every new position with value 0. We then take these modified A and B with C and can apply our reduction since the sizes are now the same. Looking through the cases $1 - 5$ of the proof, one can see that they still hold up, even if these modified positions are considered.

For the next reduction from d-MaxConv to d-MaxConv UpperBound, we will prove that we can use an algorithm for d-MaxConv UpperBound to also identify a position that violates the upper bound property.

Lemma 4.2.5. *Consider a $T(\Pi(L), d)$ algorithm for d-MaxConv UpperBound. Let A, B, C be three d-dimensional matrices of size L. We can then in time $\mathcal{O}(T(\Pi(L), d) \cdot d \cdot \log(L_{max}))$ find a position $v < L$ such that $C_v < (A \oplus B)_v$ or confirm that no such position exists.*

Proof. Without loss of generality, we will assume that a desired position $v < L$ such that $C_v < (A \oplus B)_v$ exists. We can simply apply the algorithm for d-MaxConv UpperBound and if no such v exists, then the algorithm will immediately return true and we are done.

We will argue that we can find v in the desired time via multiple binary searches. Fix a dimension $1 \leq i \leq d$ and then do a binary search as follows in order to find an index $0 \leq u_i < L_i$ such that there is a position v that we desire with $v_i = u_i$.

We first apply our algorithm to instances that have only half the amount of entries i.e. that is halved in dimension i. To be more precise, we take the sub-matrices from A, B, C each of size $L - e^{(i)} \cdot \lfloor L_i/2 \rfloor$. When the test returns false, then there is a position in the first half of the matrix and we can continue the binary search in there. Should the test return true, so for the tested part we indeed have an upper bound property, then we need to look in the half we did not test for the contradicting position. Note however that we still need to include the previously tested part to not skew the test result. See Figure 4.2 for an illustration.

To find an actual position that violates the upper bound property, we use this binary search in the first dimension to find u_1. We know that there is a conflicting position v with $v_1 = u_1$. Before we apply the same approach in the second dimension however, we modify C by setting any C_p for $p < L$ with $p_1 \neq u_1$ to a sufficiently large value, i.e. ∞.

With this modification, we will make sure that no position $p < L$ with $p_1 \neq u_1$ will cause the d-MaxConv UpperBound test to fail. This may remove potential candidates for a conflicting position v but as we found u_1, at least one position will

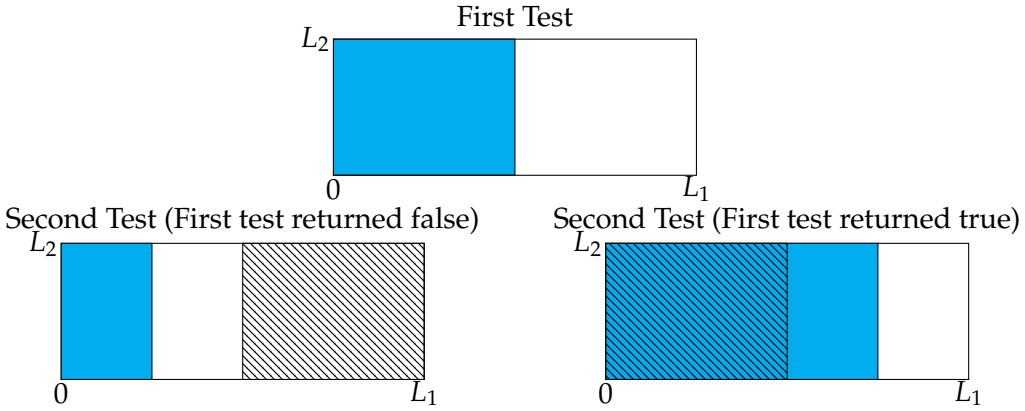


FIGURE 4.2: Sketch of the binary search. The colored area is the section from A, B, C we apply the oracle on. The hatched part shows the part, that we do not specifically test for, because we know there is a faulty position in the unhatched part.

remain. Now we can do the binary search for the second dimension and get u_2 with the conclusion that there is a conflicting position v with $v_1 = u_1$ and $v_2 = u_2$.

We repeat this process, the binary search and respective modification of C, for each dimension. By induction we can conclude, that the resulting position u is in fact a conflicting position. Each application of the algorithm takes time $T(\Pi(L), d)$. The algorithm is repeated $\mathcal{O}(\log(L_{max}))$ times for each binary search and we use d such binary searches leading to the overall running time. □

Theorem 4.2.6 (d-MAXCONV $\rightarrow$ d-MAXCONV UPPERBOUND)**.** . *A $T(\Pi(L), d)$ algorithm for d-MAXCONV UPPERBOUND implies an $\mathcal{O}(2^d \Pi(L) \cdot T(2^d \sqrt{\Pi(L)}, d) \cdot d \cdot \log(L_{max}))$ algorithm for d-MAXCONV.*

Proof. The proof consists of two parts. First, we will explain the general algorithm to find the correct convolution. The idea behind this procedure is to calculate every convolution entry via binary search. These binary searches are then processed in parallel, that is we start a matrix where each position holds a guess for that position. For the next iteration we test for each position whether its value is too low or too high and calculate for the next iteration a new full matrix with new guesses. In the second part, we will further specify how to achieve the desired running time.

The general idea is that we will do binary search for the value of $(A \oplus B)_v$. Note that we may assume that every matrix only has non-negative entries, since we can add a large enough value to all positions and subtract it later from the convolution. Let A_{max}, B_{max} be the maximal entries in A and B respectively, then $K = A_{max} + B_{max}$ is an upper bound for the convolution value in any position and we do binary search for C_v for all positions $v < L$ in the area from 0 to K.

We start by constructing a matrix $C^{(0)}$ with $C^{(0)}_v = \lfloor K/2 \rfloor$. We now want to identify for $C^{(0)}$ which entries are too small for the convolution. For that, we apply our oracle from Theorem 4.2.5 and identify a position u that is contradicting the upper bound property.

We know for this position u, that the current value is too small and we need to look for a larger value between $\lfloor K/2 \rfloor + 1$ and K in the next iteration. For the next iteration of the binary search, we will generate a new matrix $C^{(1)}$ and we can set $C^{(1)}_u$ to the next guess for the binary search. Before we continue however, we need

to find all contradicting positions in $C^{(0)}$. To identify these, we update $C_u^{(0)} = K$ and apply our oracle again. Since K is large enough, $C_u^{(0)}$ will not return as contradicting position and so our oracle will return a new contradicting position if there is one.

By repeating this process, we can identify all values in our current matrix that are too small for the convolution and set them accordingly in $C^{(1)}$. At some point, our oracle will return that all entries fulfill the upper bound property. For all the entries that were never returned by our oracle, we know their value in the maximum convolution lies between 0 and $\lfloor K/2 \rfloor$. We can therefore also set the appropriate next binary search guess in $C^{(1)}$ for these positions. We then can apply this same procedure for $C^{(1)}$ and inductively compute further matrices. After $\mathcal{O}(\log K)$ repetitions, we will have identified the convolution entries for all positions.

Just blindly applying our oracle to the whole matrix however is too slow, so we will split our matrix in sub-matrics and apply our oracle in a specific way to be more efficient. Assume for the following part that every L_i is a square number. We then split every matrix into $m = \sqrt{\Pi(L)}$ sub-matrices with up to $\sqrt{L_i}$ entries in each dimension $1 \leq i \leq d$. Enumerate these blocks or sub-matrices from 1 to m and let I_j denote the set of positions in block $1 \leq j \leq m$. We now use our oracle to test certain combinations of these sub-matrices that we call chunks.

We can test for two chunks i, j whether for every $v \in I_i$, $u \in I_j$ we have that $A_v + B_u \leq C_{v+u}$ or find positions that violate this constraint using the oracle. The combination of $v + u$ however may be contained not in only one but 2^d many chunks. We therefore apply the algorithm to matrices of size $2\sqrt{L_i}$ in each dimension. To be precise, we define A', B' to be the matrices defined by the chunks I_i and I_j and expand these to the desired size by filling them up with small dummy entries $-K$. C' is then the sub-matrix that contains all positions $v + u$ for $v \in I_i$ and $u \in I_j$.

Now we can apply our oracle in the sense of the binary search procedure that we described above. Assume we already have one matrix $C^{(i)}$ and combine every possible combination of chunks. One run of the oracle based on Theorem 4.2.5 takes time $\mathcal{O}(T(2^d\sqrt{\Pi(L)}, d) \cdot d \cdot \log(\sqrt{L_{max}}))$, since we expanded the size of each chunk to gain equal sized matrices. We repeat this test for all possible combinations of chunks, which are in total $m^2 = \Pi(L)$.

Further, if a chunk yields a contradicting position, we update the value in $C^{(i+1)}$ and we repeat the test of the chunk again. However, by setting the faulty position with a large value each position can only contradict the upper bound property once and therefore the necessary number of repetitions for this is again bounded by $\Pi(L)$. In total, we can upper bound the running time by $\mathcal{O}(\Pi(L) \cdot T(2^d\sqrt{\Pi(L)}, d) \cdot d \cdot \log(\sqrt{L_{max}}))$. Remember that we omitted the factor of $\mathcal{O}(\log K)$ for the overall binary search on each position. Since by definition of K we have $K \leq 2W$ the desired result follows.

Note that if L_i is not a square number, we might end up with additional incomplete sub-matrices that we need to combine. This might incur another factor of 2^d for the number of matrices that have to be combined. As d is fixed, this will also not alter the running time further.

□

The last reduction is based on Cygan et al. [36] and Bringmann [21]. To make their approach work even in higher dimensional cases, we require a new more refined distribution of items into so-called layers. With this new partition of items many used techniques such as Color-Coding or merging partial solutions naturally translate into higher dimensions. We give a full detailed analysis and the algorithms in section 4.3.

Theorem 4.2.7 ($0/1$ d-KNAPSACK $\rightarrow$ d-MAXCONV)**.** *If d-MAXCONV can be solved in time $T(\Pi(L), d)$ then $0/1$ d-KNAPSACK can be solved with a probability of at least $1-\delta$ in time $\mathcal{O}(T(\Pi(12t), d)\log(d \cdot t_{max})\log^3(\log n/\delta) \cdot d \cdot \log n)$ for any $\delta \in (0,1)$.*

We remark that this also yields a respective algorithm for $0/1$ d-KNAPSACK with the trivial $\Pi(L)^2$ algorithm for d-MAXCONV. Whether further lower order improvements like the results Bremner [20] or Chan and Lewenstein [28] is still open.

4.3 Reduction from Knapsack to Convolution

In this section, we present a reduction from $0/1$ d-KNAPSACK to d-MAXCONV. This reduction will be given through an algorithm that solves $0/1$ d-KNAPSACK by computing multiple convolutions. In fact, we will solve the proposed Knapsack problems for all possible capacities $t' \leq t$.

The solution we give is then a matrix A, call it *knapsack matrix* in the following, of size $t + \overrightarrow{\mathbf{1}}$ such that $A_{t'}$ denotes the maximum profit of a knapsack with capacity $t' \leq t$. Our algorithm follows a basic idea: We randomly split the items from the instance into different subsets, compute solution matrices for these subsets and then combine them using convolution.

We note that these matrices have size $t + \overrightarrow{\mathbf{1}}$ and that combining matrices of this size takes time $T(\Pi(t + \overrightarrow{\mathbf{1}}), d) \in \mathcal{O}(2^d T(\Pi(t), d))$. When d is fixed, the running time would be $\mathcal{O}(T(\Pi(t), d))$. For this reason, we will omit that the knapsack matrix we compute is technically slightly larger than t and still write that computing convolutions of this sizes takes time $\mathcal{O}(T(\Pi(t), d))$.

The randomized algorithm we present for this reduction is a simple variation of the one introduced by Cygan et al. [36] and Bringmann [21]. The trick is to choose specific subsets of items, solve the Knapsack problem for these sizes and then combine the solutions via convolution. As such, it is important to decide how to build fitting sub-instances in the higher dimensional case.

In the following we will present these algorithms and prove their correctness. We start off with a randomized algorithm that will find an optimal solution with a bounded number of items via a technique called ColorCoding [21]. (see also Listing 4.1).

Lemma 4.3.1. *For any Knapsack instance with item set I and capacity t, denote with $\Omega(I)_v$ the maximum profit of a subset $I' \subset I$ with total weight smaller or equal to $v \leq t$.*

Let $\delta \in (0,1)$ and $k \in \mathbb{N}$. There exists an algorithm that for every Knapsack item set I and capacity t computes a random matrix S in time $\mathcal{O}(T(\Pi(t), d)k^2 \log(1/\delta))$ such that for any $Y \subseteq I$ with $|Y| \leq k$ and every capacity $v \leq t$ we have $\Omega(Y)_v \leq S_v \leq \Omega(I)_v$ with probability $\geq 1-\delta$.

Proof. We initialise k^2 matrices $Z^{(1)}, \cdots, Z^{(k^2)}$ of size $t + \overrightarrow{\mathbf{1}}$ and randomly distribute items from I into $Z^{(1)}, \cdots, Z^{(k^2)}$. The probability for an item to land in either matrix is equally distributed. When an item with weight v and profit p is assigned to $Z^{(i)}$, we set $Z^{(i)}_v := p$. If two items of the same weight are assigned to one matrix, we keep the highest profit. Ultimately we want that items from an optimal solution are split among our matrices.

Take an optimal solution Y with at most k elements. The probability that all these items are split in different $Z^{(i)}$ can be bounded by:

$$\frac{k^2-1}{k^2} \cdot \frac{k^2-2}{k^2} \cdots \frac{k^2-(|Y|-1)}{k^2} \geq (1-\frac{k-1}{k^2})^k \geq (1-\frac{1}{k})^k \geq (\frac{1}{2})^2$$

So with probability of at least $1/4$, all elements of Y are split among the k^2 matrices, that is no two items in Y are in the same matrix. We build the convolution of all $Z^{(i)}$'s and get S as a result in running time $\mathcal{O}(T(\Pi(t,d)k^2)$. By repeating this process $\mathcal{O}(\log(1/\delta))$ times, we can guarantee the desired probability, while achieving the proposed total running time. □

LISTING 4.1: ColorCoding

```
ColorCoding(I, t, k, δ)
for j ∈ {1, ··· , ⌈log_{4/3}(1/δ⌉}
    randomly partition I = Z^(1) ∪ ··· ∪ Z^(k^2)
    compute R^(j) = Z^(1) ⊕ ··· ⊕ Z^(k^2)
end for
for all v ≤ t set S_v := max_j P_v^(j)
return S
```

We now split a given item set I with $n = |I|$ into disjoint layers based on the capacity of the knapsack t. Denote with L_j^i the set of items with a weight vector w such that $w_i \in (t_i/2^j, t_i/2^{j-1}]$ for $i < d$ and $j < \lceil \log n \rceil$. Denote with $L^i_{\lceil \log n \rceil}$ the last layer holding items whose weight vector has that $w_i \leq t_i/2^{\lceil \log n \rceil - 1}$.

Now every item appears in d different layers. Choose therefore for an item one layer L_j^i that contains it and such that j is minimal. If there are multiple possibilities, choose arbitrarily among these and remove said item from any other layers. We note that from a layer L_j^i, we can choose at most 2^{j-1} items for a knapsack solution as otherwise the i-th capacity constraint would be exceeded. By choice of layers, it is also possible for any solution to contain that many items. As we have chosen j minimal, there can be no other weight constraint invalidating our solution.

In the following lemma, we will now prove how to solve one of these layers. The algorithm for that can be seen in Listing 4.2.

LISTING 4.2: Algorithm to solve one layer using ColorCoding

```
SolveLayer(L, t, j, δ)
Set l = 2^j, γ = 6 log(l/δ)
Let m = l/log(l/δ) be rounded to the next power of 2.
if l < log(l/δ) then return ColorCoding(L, t, l, δ)
randomly partition L = A_1 ∪ ··· ∪ A_m
for k ∈ [m]
    R^(j) = ColorCoding(A_k, (2γ/l)t, γ, δ/l)
for h ∈ [log m]]
    for k ∈ m/(2^h)
        P_k = P_{2k-1} ⊕ P_{2k}
    end for
end for
return P_1
```

Lemma 4.3.2. *For all layers L_j^i and $\delta \in (0, 1/4]$ there exists an algorithm that computes a matrix W in time $\mathcal{O}(T(12\Pi(t), d)\log(d \cdot t_{max})\log^3(2^j/\delta))$, where for each capacity $v \leq t$ we have that $W_v = \Omega(L_j^i)_v$ with a probability of at least $1 - \delta$.*

Proof. Consider a layer L_j^i and set the following parameters based on the algorithm in Listing 4.2: $l = 2^j$, $m = l/\log(l/\delta)$ rounded up to the next power of 2 and $\gamma = 6\log(l/\delta)$.

Consider the case where $l < \log(l/\delta)$, we then compute the desired result in time $\mathcal{O}(T(\Pi(t), d)l^2 \log(1/\delta)) \in \mathcal{O}(T(\Pi(t), d)\log^3(l/\delta))$. We do this by applying Theorem 4.3.1 with probability parameter δ and bound on the number of items l.

In the general case where $l \geq \log(l/\delta)$, we split the item set into m disjoint subsets $A_1, \cdots, A_m$. We apply again Theorem 4.3.1. We use δ/l for the probability parameter and allow γ items for each subset. We set the capacity to $\frac{2\gamma}{l} \cdot t$, which is the maximum weight of any packing that uses at most γ items with size at most 2^{j-1}, which is the maximum size in layer L_j^i. To calculate a solution matrix of A_k via algorithm Listing 4.1, we need time:

$$\mathcal{O}(T(\Pi(2\gamma/l \cdot t), d)\gamma^2 \log(l/\delta)) \in \mathcal{O}(T(\Pi(12\log(l/\delta)/l \cdot t), d)\log^3(l/\delta)).$$

We want to note two things for our estimation. First, for $x > 0$ we have that $T(x, d) \in \Omega(x)$, so for $y \geq 1$ we have that $y \cdot T(x, d) \leq (T(y \cdot x, d)$. For the same reason, we have $y \cdot \Pi(v) \leq \Pi(y \cdot v)$ for some vector v with $v \geq \overrightarrow{\mathbf{1}}$. Denote with $m' = l/\log(l/\delta)$ the value of m before rounding and conclude that $2m' \geq m$.

Together to solve all A_k we therefore require in total:

$$\begin{aligned}\mathcal{O}(m'T(\Pi(12\log(l/\delta)/l \cdot t), d)\log^3(l/\delta)) &\in \mathcal{O}(T(\Pi(m' \cdot 12\log(l/\delta)/l \cdot t), d)\log^3(l/\delta)) \\ &\in \mathcal{O}(T(\Pi(12 \cdot t), d)\log^3(l/\delta))\end{aligned}$$

We now need to combine all the A_k and we do so in a binary tree fashion. In round $h \in [\log m]$, we have $m/2^h$ convolutions, each with a result matrix of size $2 \cdot 2^h\gamma/l \cdot t$. The convolutions take total time:

$$\begin{aligned}\sum_{h=1}^{\log m} \frac{m}{2^h} T(\Pi(2 \cdot 2^h\gamma/l \cdot t), d) &\leq \sum_{h=1}^{\log m} 2 \cdot \frac{m'}{2^h} T(\Pi(2^h \cdot 12\log(l/\delta)/l \cdot t), d) \\ &\leq \sum_{h=1}^{\log m} 2 \cdot T(\Pi(\frac{m'}{2^h} \cdot 2^h \cdot 12\log(l/\delta)/l \cdot t), d) \\ &= \sum_{h=1}^{\log m} 2 \cdot T(\Pi(12 \cdot t), d) \in \mathcal{O}(T(\Pi(12t), d)\log m)\end{aligned}$$

We might require arithmetic overhead due to the large values of profits. A knapsack with size t may only hold $||t||_1$ items whose weight is not $\overrightarrow{\mathbf{0}}$. Note that we may remove items with weight $\overrightarrow{\mathbf{0}}$ without loss of generality and we can conclude that the maximum profit is bounded by $||t||_1 \cdot W \in \mathcal{O}(d \cdot t_{max}W)$. We add a factor of $\log(d \cdot t_{max})$ to the runtime to compensate for the arithmetic overhead.

All that is left is to argue that our algorithm works correctly and with the proposed probability. We note that the first case follows immediately from the Theorem 4.3.1. For the general case fix an optimal item set Y and set $Y_k = Y \cap A_k$ for $k \in [m]$. Based on the distribution of items $|Y_k|$ is distributed as the sum of $|Y|$ independent Bernoulli random variables. Each variable has a success probability of $1/m$ and the expected value is $\mu := \mathbb{E}[|Y_k|] = |Y|/m \leq l/m \leq \log(l/\delta)$. According to a standard Chernoff bound, it holds that $Pr[|Y_k| \geq \lambda] \leq 2^{-\lambda}$ for any $\lambda \geq 2e \cdot \mu$. By

using $\gamma = 6\mu \geq 2e \cdot \mu$, we get that $Pr[|Y_k| \geq \gamma] \leq 2^{-\gamma} = \frac{1}{2^\gamma} \leq \frac{1}{2^{\log(l/\delta)}} = \frac{\delta}{l}$ and this describes the probability that some partition A_k has too many elements from Y.

With a probability of each at least $1 - \frac{\delta}{l}$, we have that $|Y_k| \leq \gamma$ and therefore our algorithm for Theorem 4.3.1 is applicable to find the solution generated by Y_k or one of equal value for the same weight. The probability that this happens for all partitions is at least $(1 - \frac{\delta}{l})^m \geq 1 - m\frac{\delta}{l}$ using the Bernoulli inequality. The probability that we find the solution generated by Y_k based on Theorem 4.3.1 is also at least $1 - \frac{\delta}{l}$, so the probability that this works also for all $i \in [m]$ is $\geq 1 - m\frac{\delta}{l}$. The probability of both events, the proper split and finding the right solution is then at least $(1 - m\frac{\delta}{l})^2 \geq 1 - 2 \cdot m\frac{\delta}{l}$. We can further note that $m \leq l/2$ and therefore $1 - 2 \cdot m\frac{\delta}{l} \geq 1 - \delta$.

□

Now that we can solve each layer, all that remains is to combine the solutions for each layer. To do so, we simply apply convolution again.

Proof of Theorem 4.2.7. As discussed earlier, we split the set of items into the respective layers and solve each layer via Theorem 4.3.2. We use as probability parameter $\delta' := \delta/\lceil\log(n)\rceil$. To solve a layer, we require time $\mathcal{O}(T(12\Pi(t), d)\log(d \cdot t_{max})\log^3(\log n/\delta))$ and we do this for all $d \cdot \lceil\log n\rceil$ layers. We then use the convolution algorithm to combine the resulting matrices from each layer in time $\mathcal{O}(T(\Pi(t), d) \cdot d \cdot \log n)$. Putting all this together, we require a total time in $\mathcal{O}(T(12\Pi(t), d)\log(d \cdot t_{max})\log^3(\log n/\delta) \cdot d \cdot \log n)$.

For the probability, consider some optimal solution Y and $Y_j^i := L_j^i \cap Y$. Based on Theorem 4.3.2, we find Y_j^i or a solution of equal profit for the same weight with a probability of $\geq 1 - \delta' = 1 - \delta/\lceil\log(n)\rceil$. The chance that we find the proper solutions over all layers is then at least $(1 - \delta/\lceil\log(n)\rceil)^{\lceil\log n\rceil} \geq 1 - \delta$ with the Bernoulli inequality.

□

4.4 Parameterized Algorithm for Multi-Dimensional Knapsack

In this part, we will consider solving Knapsack with only completely filled packings. This will make some of the following formulations simpler and easier to understand.

0/1 d-EQUALITYKNAPSACK	
Input:	Set I of n items each defined by a profit $p_i \in \mathbb{R}_{\geq 0}$ and weight vector $w^{(i)} \in \mathbb{N}^d$ along with a knapsack of capacity $t \in \mathbb{N}^d$.
Problem:	Find subset $S \subseteq I$ such that $\sum_{i \in S} w^{(i)} = t$ and $\sum_{i \in S} p_i$ is maximal.

We will also solve this problem not only for the given capacity t but for all capacities. In detail, we will construct a solution matrix A with size $t + \overrightarrow{\mathbf{1}}$ such that for any $v \leq t$ we have that A_v is the maximum profit of an item set whose total weight is exactly v. One can observe that we can construct a solution for 0/1 d-KNAPSACK after solving 0/1 d-EQUALITYKNAPSACK by simply remembering the highest profit achieved in any position.

The algorithm we will show is based on the algorithm of Axiotis and Tzamos [2]. They solved 1-dimensional Knapsack by splitting all items into subsets with equal weight. They proceed then to solve these resulting sub-instances. These sub-instances can be easily solved by gathering the highest profit items that fit into the

knapsack. Finally they combine these resulting partial solutions via Convolution. The profits of one such partial solution build a concave sequence, which allows each convolution to be calculated in linear time.

Definition 4.4.1 (Concave Sequences)**.** Let $a = (a_i)_{i \leq n}$ be a sequence of numbers of length n. We call *a concave* if for all $i \leq n-2$ we have $a_i - a_{i+1} \geq a_{i+1} - a_{i+2}$.

We will achieve a similar result and calculate higher dimensional convolutions in linear time. In fact we will reduce our problem to calculating convolution of 1-dimensional concave sequences. This way we can calculate the convolution of our sub-solutions in linear time in the number of entries in our matrix. For simplicity, we will again assume d to be fixed. Removing this assumption, we might need to incur another additional factor of d for the final running time.

Theorem 4.4.1 (Restatement of Theorem 4.1.2 for fixed dimension)**.** *Let d be a fixed constant. For every $0/1$ d-*EQUALITYKNAPSACK *instance with D different weight vectors and capacity t, let $V := \Pi(t + \overrightarrow{\mathbf{1}})$ be the number of possible capacities $v \leq t$. Then we can solve the given instance in time $\mathcal{O}(n + D \max\{V, t_{\max} \log t_{\max}\})$.*

We remark, that in most cases $V > t_{\max} \log t_{\max}$ so generally this algorithm will have a running time of $\mathcal{O}(n + DV)$. However, in the special case of a slim matrix that has size t with one large value t_j and other values being very small $t_i << t_j$ for $i \neq j$ our algorithm might have a slightly worse running time of $\mathcal{O}(n + Dt_{\max} \log t_{\max})$.

4.4.1 Algorithm and Correctness

Let $W = \{w^{(1)}, w^{(2)}, \cdots, w^{(D)}\}$ denote the set of all distinct item weight vectors that are contained in the instance. The algorithm then works as follows: For every $w \in W$, partition all items into subsets $I^{(w)}$ containing all items that have weight w. Sort the items of each partition nonincreasing by their profit and compute a solution matrix $A^{(w)}$, where $A^{(w)}_v$ is the maximum profit for a knapsack of capacity $v = t$ using only items in $I^{(w)}$.

These solutions can be easily computed, as $A^{(w)}_{k \cdot w}$ for any $k \in \mathbb{N}$ is given through the sum of k highest profits in $I^{(w)}$. We can set these positions accordingly. For any position $v \leq t$ that is not a multiple of w, we simply set $A^{(w)}_v = -\infty$. Remark that we also set $A^{(w)}_{k \cdot w} = -\infty$ in case there are no k items in $I^{(w)}$ to fill the knapsack and $A^{(w)}_{0 \cdot w} = A^{(w)}_{\overrightarrow{\mathbf{0}}} := 0$, as an empty packing hits exactly capacity $\overrightarrow{\mathbf{0}}$.

For simplicity, we denote $A^{(w^{(i)})}$ with $A^{(i)}$. After calculating all $A^{(i)}$, we then want to combine all these solutions. For that, we set $R^{(1)} := A^{(1)}$ and compute for all $1 < i \leq D$ the convolution $R^{(i)} := R^{(i-1)} \oplus A^{(i)}$. The solution of the overall problem is then given by $R^{(D)}_t$. The correctness of this algorithm follows from the next lemma.

Lemma 4.4.2. *Let I_1, I_2 be two item-disjoint $0/1$ d-*EQUALITYKNAPSACK *instances for the same capacity t and $R^{(1)}, R^{(2)}$ be the solution matrices for the respective instances. Consider the combined instances $I = I_1 \cup I_2$ and a respective solution matrix R for I.*

Let t' be any capacity feasible for all three instances and solution matrices, then we have that $R_{t'} = (R^{(1)} \oplus R^{(2)})_{t'}$.

Proof. Generally, we can conclude that each profit noted down in a solution matrix consists of a respective item set. Let $I' \subset I$ be the set of items that make up the maximum profit $R_{t'}$ for instance I.

Let $I'_1 \cup I'_2 = I'$ be the partition with $I'_i \subseteq I_i$ for $i \in \{1,2\}$ and denote with $v^{(i)}$ the total weight of I'_i. Note that each I'_i also has to be optimal for capacity $v^{(i)}$ in instance I_i as otherwise we could improve I'_i and I', respectively.

Assume that $R_{t'} < (R^{(1)} \oplus R^{(2)})_{t'}$, then this would imply that there are two better solutions from I_1 and I_2 that make for a higher profit solution for I when combined, however $R_{t'}$ is already maximal.

Assume that $R_{t'} > (R^{(1)} \oplus R^{(2)})_{t'}$. Note that $v^{(1)} + v^{(2)} = t'$ and hence $v^{(1)} = t' - v^{(2)}$. We then have in total another contradiction with:

$$R_{t'} = R^{(1)}_{v^{(1)}} + R^{(2)}_{v^{(2)}} \leq R^{(1)}_{t'-v^{(2)}} + R^{(2)}_{v^{(2)}} \leq (R^{(1)} \oplus R^{(2)})_{t'}$$

□

We remark that all the initial solutions $A^{(i)}$ are optimal, as the solution contain the highest profit items. For that reason exchanging items from the initial solutions, will not change the weight but can at best decrease the profit. Therefore, applying this lemma inductively implies the optimality of the final solution. As we will see in the following part, we do not need the actual matrix $A^{(i)}$ but the non-infinity values of it will suffice to calculate the respective convolutions.

Lemma 4.4.3. *Given an item set I, we can partition I into subsets $I^{w^{(i)}}$ and calculate all values $A^{(i)}_{w^{(i)}}, A^{(i)}_{2w^{(i)}}, \cdots$, for all $i \in [D]$ in time $\mathcal{O}(n + D \cdot t_{\max} \log(t_{\max}))$.*

Proof. The first part of partitioning the items into the respective subsets can be done by iterating through all items and putting them in a respective subset $I^{(w^i)}$ based on their weight vector. This can be simply done in $\mathcal{O}(n)$.

Set $n_i := |I^{w^i}|$ and let k_i be the number of items of $I^{w^{(i)}}$ that fit into the knapsack. In order to calculate the values $A^{(i)}_{w^{(i)}}$, all we need to do is finding the k_i highest profit items in I^{w^i}. In linear time $\mathcal{O}(n_i)$ we can find the k_i-th highest profit p in and by simply iterating over I^{w^i} we can omit all items with less profit and if necessary also remove items with profit p until we only have k_i items. We can then sort these items in time $\mathcal{O}(k_i \log k_i) \in \mathcal{O}(t_{\max} \log t_{\max})$.

If we do this for all item groups, then we end up with a total running time of

$$\mathcal{O}(\sum_{i=1}^{D}(n_i + t_{\max} \log t_{\max})) = \mathcal{O}(n + D \cdot t_{\max} \log t_{\max})$$

□

4.4.2 Computing the Convolution

To achieve the proposed running time, we need to calculate convolutions fast. In order to do so, we will also use the concept of concave sequences.

We note that the solutions of our sub-instances make up concave profit sequences given by $A^{(i)}_{0 \cdot w^{(i)}}, A^{(i)}_{1 \cdot w^{(i)}}, A^{(i)}_{2 \cdot w^{(i)}} \cdots$. Further, we will show that we can now calculate the d-dimensional convolution of $R^{(i)} = R^{(i-1)} \oplus A^{(i)}$ by computing a multitude of 1-dimensional convolutions on sequences. Before we begin, we want to make a small arrangement about the notation.

Remark 5. Let A be a d-dimensional matrix of size L and $v \in \mathbb{Z}^d$ be an *invalid position* of A, that is either $v_i < 0$ or $v_i \geq L$ for some $1 \leq i \leq d$. We then define $A_v := -\infty$.

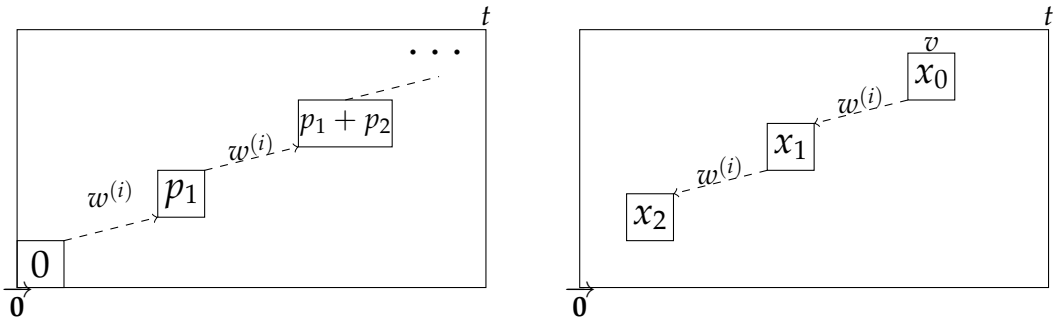


FIGURE 4.3: The left picture shows the general structure of a matrix $A^{(i)}$ when the highest profits are $p_1 \geq p_2 \geq \cdots$. The value in the matrix starts with 0 and always changes at the positions $w^{(i)}, 2w^{(1)}, 3w^{(1)}, \cdots$. The right picture shows a matrix $R^{(i-1)}$ and the entries/positions that match against values from $A^{(i)}$, when computing the convolution at position v.

We will work with multiple convolutions and we will use this notation to avoid introducing confusing bound-check variables. With this, if some entry in the convolution hits an undefined position it will default to $-\infty$ and therefore not be regarded in the convolution. By construction our $A^{(i)}$ matrix has not many different values. In the next lemma, we argue that not many different values from the other matrix are needed in order to calculate the d-dimensional convolution.

Lemma 4.4.4. *Let $i \geq 2$ and consider $R^{(i)}, R^{(i-1)}, A^{(i)}$ as per the algorithm and let k be the number of unique non-zero entries in $A^{(i)}$ that are not $-\infty$. Then for every capacity $v \leq t$ we have $R^{(i)}_v = \max_{0 \leq l \leq k}(R^{(i-1)}_{v-l \cdot w^{(i)}} + A^{(i)}_{l \cdot w^{(i)}})$.*

Proof. By construction, of $A^{(i)}$ many positions default to $-\infty$ and these positions will not matter for the actual convolution. Consider position $v \leq t$, then we have by definition that:

$$R^{(i)}_v = (R^{(i-1)} \oplus A^{(i)})_v = \max_{0 \leq u \leq v}(R^{(i-1)}_u + A^{(i)}_{v-u}) = \max_{0 \leq u \leq v}(R^{(i-1)}_{v-u} + A^{(i)}_u)$$

We can reduce this formula to those positions where $A^{(i)}$ is not $-\infty$. These are exactly the values $A^{(i)}_{j \cdot w^{(i)}}$ for $0 \leq l \leq k$. We then have:

$$\max_{0 \leq u \leq v}(R^{(i-1)}_{v-u} + A^{(i)}_u) = \max_{0 \leq l \leq k}(R^{(i-1)}_{v-l \cdot w^{(i)}} + A^{(i)}_{l \cdot w^{(i)}})$$

□

We see that for one position v calculating $R^{(i)}_v$ only requires specific entries, which are depicted in Figure 4.3. We show next that, we can calculate multiple entries from the d-dimensional convolution matrix $R^{(i)}$ by constructing a sequence from $R^{(i-1)}$ and building the 1-dimensional convolution of this sequence and the sequence of unique values in $A^{(i)}$.

Lemma 4.4.5. *Consider $i \geq 2$ and $R^{(i)}, R^{(i-1)}, A^{(i)}$ as per the algorithm and let $v \leq t$ be a position with $v_j < w^{(i)}_j$ for some $j \in [d]$. Further, let $m \in \mathbb{N}$ be maximal such that $v + mw^{(i)} \leq t$ still holds. Construct now the sequences defined by $x^{(v)}_j := R^{(i-1)}_{v+jw^{(i)}}$ and $y_j := A^{(i)}_{jw^{(i)}}$ for $0 \leq j \leq m$.*

We then have for any $0 \leq l \leq m$ *that* $R^{(i)}_{v+lw^{(i)}} = (x^{(v)} \oplus y)_l$.

Proof. Fix v and write for simplicity $x := x^{(v)}$. By definition it follows that $(x \oplus y)_l = \max_{0\leq j\leq l}(x_{l-j} + y_j) = \max_{0\leq j\leq l}(R^{(i-1)}_{v+(l-j)w^{(i)}} + A^{(i)}_{jw^{(i)}})$.

By Theorem 4.4.4 we know further that for a respective k
$R^{(i)}_{v+lw^{(i)}} = \max_{0\leq j\leq k}(R^{(i-1)}_{(v+lw^{(i)})-j\cdot w^{(i)}} + A^{(i)}_{j\cdot w^{(i)}}) = \max_{0\leq j\leq k}(R^{(i-1)}_{v+(l-j)w^{(i)}} + A^{(i)}_{j\cdot w^{(i)}})$.

Note that for $j > l$, we have $R^{(i-1)}_{v+(l-j)w^{(i)}} = -\infty$ since $v_j < w^{(i)}_j$ and therefore $v + (l-j)w^{(i)}$ denotes an invalid position. Further if $l > k$ then for any $j > k$ we have $A^{(i)}_{j\cdot w^{(i)}} = -\infty$ as there are no l items to fill the knapsack. Therefore the equality $(x \oplus y)_l = R^{(i)}_{v+lw^{(i)}}$ follows. □

We note the sequence we generate from $A^{(i)}$ is concave and therefore we have the same setting as in Axiotis and Tzamos [2]. Hence we can calculate this convolution in linear time.

Lemma 4.4.6. *Let* $i \geq 2$ *and consider* $R^{(i-1)}, A^{(i)}$ *as per the algorithm. We can compute* $R^{(i)}$ *in time* $\mathcal{O}(d \cdot \Pi(t))$.

Proof. Let k be the number of items from $I^{(w^{(i)})}$ that fit into a knapsack of capacity t. Note that $A^{(i)}$ then holds $k+1$ values. From Theorem 4.4.5, we know that it is sufficient to compute convolutions of sequences $x^{(v)}, y$, both of length $k+1$ for multiple positions $v \leq t$. We do this for every position v where for some $j \in [d]$ we have $v_j < w^{(i)}_j$. We now want to prove there are at most $d \cdot \Pi(t)/k$ such positions.

Define with $P_j := \{v \leq t | v_j \leq t_j/k\}$ the set of positions that are bounded in the j-th dimension by t_j/k. To prove the overall statement, we want to show that $w^{(i)}_j \leq t_j/k$ for all $j \in [d]$ and therefore the set of positions we calculate convolutions for is contained in $\bigcup_{j\in[d]} P_j$. We then can bound the number of convolutions we build by $\sum_{j\in[d]} |P_j| \leq \sum_{j\in[d]} \Pi(t)/k = d \cdot \Pi(t)/k$ and the statement follows.

The fact that $w^{(i)}_j \leq t_j/k$ holds can be quickly seen via an indirect proof. Assume for the sake of contradiction that we have $w^{(i)}_j > t_j/k$ for some j. Remember that k was the number of items with weight $w^{(i)}$ that the given knapsack can hold. However, we now have $k \cdot w^{(i)}_j > t_j$ and therefore this is not possible anymore.

The running time now results by applying the result by Axiotis and Tzamos. Every convolution itself can be computed in time $\mathcal{O}(k)$ and this is done for all the necessary positions, which is at most $d \cdot \Pi(t)/k$. □

Ultimately, we need to calculate $D-1$ such convolutions and that will prove Theorem 4.4.1.

4.5 Generalisation to ILPs

4.5.1 Negative Weights and ILP

As we mentioned before, Knapsack is a special case of ILP. We now want to discuss how to extend our results to ILPs.

Lemma 4.5.1. *Solving* $0/1$ d-EQUALITYKNAPSACK *is equivalent to solving the ILP problem defined in section 1 for a matrix* $A \in \mathbb{Z}^{d\times n}$, *where every entry in* A *is non-negative and* $u = \overrightarrow{\mathbf{1}}$.

Proof. For every knapsack item, we add a column to the matrix A. For the i-th item with weight vector $w^{(i)}$, we set the i-th column of A to $a^{(i)} := w^{(i)}$ and set the respective cost vector to $c_i := p_i$ to the profit of the item. The target vector we set to $b := t$ and the upper bound vector $u := \overrightarrow{\mathbf{1}}$.

Given a solution for the Knapsack problem, that is an item collection I', we then set $x_i = 1$ for every $i \in I'$ and $x_i = 0$ for every $i \notin I'$. We note that this reduction can be reversed and an ILP instance can be turned into a Knapsack instance respectively. In the same way solutions can be transformed. The equivalence then follows from the fact, that the profits of a solution is equal to the profit of the transformed solution. □

Note that we are also able to handle Knapsack instances with negative item profits. In the equality version, this makes sense because we might be forced to take a negative profit item to hit the capacity vector. Further if multiple items with equal weight and profit exist, we can summarize them into one column and set u_i to the number of occurrences of item i and delete every copy to not count them more than intended.

For the next part, we allow negative weight values, so we have for every item that $w^{(i)} \in [-\Delta, \Delta]^d$. We will use this Δ again to parameterize the algorithm. We note that this problem resembles the general problem of solving an ILP where each entry in the constraint matrix is bounded by Δ. In order to apply our previous algorithm, we will reduce the instance of this d-dimensional problem to a $(d+1)$-dimensional problem with no negative weights.

The reduction goes as follows: For every item i with weight vector $w^{(i)}$, we introduce a new item with weight $w'^{(i)}$ where $w_j'^{(i)} = w_j^{(i)} + \Delta$ for $j \in [d]$ and $w_{d+1}'^{(i)} = 1$. We do not change the profit of the items. As for the capacity, we will guess the number of items in the optimal solution. Let K be such a guess and t be the original capacity. We define the respective instance for this guess through the capacity $t^{(K)}$ with $t_j^{(K)} := t_j + K\Delta$ for $j \in [d]$ and $t_{d+1}^{(K)} := K$.

Corollary. *There is an algorithm that finds an optimal solution with exactly $K \in \mathbb{N}$ items to $0/1$ d-*EQUALITYKNAPSACK *with weight entries bounded by Δ in absolute value in time $\mathcal{O}(n + D \cdot K \cdot \Pi(t + \overrightarrow{\mathbf{K\Delta}})) \subseteq \mathcal{O}(n + D \cdot K \cdot (t_{\max} + K\Delta)^d)$.*

Proof. We apply Theorem 4.1.2 to the modified instance we mentioned above. We note that the number of different items does not change at all, since all we do is add a 1 in the $(d+1)$-th component and all other components are modified in the same way. Further, by construction, we get $\Pi(t^{(K)}) = K \cdot \Pi(t + \overrightarrow{\mathbf{K\Delta}})$. Using this in our previous algorithm we get the desired result.

Lastly, we also want to prove the correctness of the reduction. Denote with I the original item set, where items may have negative weight and with I' the transformed item set as per the reduction mentioned above. Let $S' \subset I'$ be an optimal solution to the modified instance and let S be the set of unmodified items that are in S'. Based on the $(d+1)$-th weight we added, we know that $|S| = |S'| = K$.

Since S' is a solution for the modified version, we have $\sum_{i \in S'} w_j'^{(i)} = t_j + (\overrightarrow{\mathbf{\Delta K}})_j$ for all $j \in [d]$. By definition, $w_j'^{(i)} = w_j^{(i)} + \Delta$ and since S' and S hold K items, we in total have:

$$\sum_{i \in S} w_j^{(i)} = \sum_{i \in S'} w_j'^{(i)} - \Delta K = t_j.$$

Finally, we can also note that the profit between S and S' does not change and therefore S must also be optimal. If there was a higher profit packing for I, we could

transform that into a better packing for I' and that would contradict the choice of I'. □

We can already apply Theorem 4.5.1 to give an algorithm for $0/1$ d-EQUALITYKNAPSACK with negative weights, by either guessing the correct value of K or enumerating over all possible $K \in [n]$. The problem is that K can grow very large, that is up to n. With a reduction introduced by Eisenbrand and Weismantel [41], we will reduce the number of possible K. The trick to make this happen is proximity.

Eisenbrand and Weismantel calculate a non-integral optimal solution x^* for the LP-relaxation, which is possible in polynomial time. Eisenbrand and Weismantel proved that there exists an integral optimal solution z^*, such that the distance in 1-norm is small i.e. $||z^* - x^*||_1 \leq d \cdot (2d\Delta + 1)^d$. They then proceed to take $\lfloor x^* \rfloor$ as an integral solution and construct an ILP to find a solution $y \in \mathbb{Z}^d$ such that $Ay = A(x^* - \lfloor x^* \rfloor)$. Through proximity and rounding, they can limit feasible solutions to y that fulfill $||y||_1 \leq d \cdot (2d\Delta + 1)^d + d =: L$. Applying this reduction gives us a bound for K and yields the following result.

Lemma 4.5.2. *There is an algorithm that solves the ILP problem in time $\mathcal{O}(n + 2D \cdot L^2 \cdot \Pi(\overrightarrow{\mathbf{d\Delta}} + \overrightarrow{\mathbf{L\Delta}})) \subseteq \mathcal{O}(n + 2D \cdot L^2 \cdot (d\Delta + L\Delta)^d)$, when an optimal solution for the LP-relaxation is given. Rewritten for fixed d this results in a time of $n + 2D \cdot \mathcal{O}(\Delta)^{(d+1)^2} \cdot \mathcal{O}(d)^{d(d+2)}$.*

Proof. Let A, x, b, c, u be given as per the problem and let x^* be an optimal solution of the relaxation. We construct the same ILP to solve the ILP constructed by Eisenbrand and Weismantel.

We note that the number of unique columns does not change. However we need to allow negatives of a column to be taken, which is not something that we worked with so far. We however modify the instance by adding negative multiples of columns that are necessary. By doing so, we may increase the number of unique columns by a factor of 2. We further note that the new target vector fulfills $A(x^* - \lfloor x^* \rfloor) \leq \overrightarrow{\mathbf{d\Delta}}$, as x^* has at most d fractional components.

This modified ILP is then transformed to a $0/1$ d-EQUALITYKNAPSACK instance. To do so, we generate an item for each column of ILP and the target vector defines the capacity of the knapsack. We then apply Theorem 4.5.1 on this instance. Since we know that an optimal solution with $||y||_1 \leq L$ exists, we can apply Theorem 4.5.1 for all $K \leq L$, which is sufficient. □

4.6 Conclusion

In total, we have shown two things. First of all, we have proven that the relationship between Knapsack and Convolution in the 1-dimensional case also prevails in higher dimensional variants. A natural follow up question is how many more problems can we entangle into this problem class.

We further have shown that strong parameterized results can be achieved as well and that knapsack in higher dimensions can be reduced to 1-dimensional convolution. It remains open whether newer improvements on algorithms on both knapsack and convolution such as [22, 23, 31] can be also extended to these higher dimensional problems.

We also wonder how far the connection between knapsack and ILPs can be taken in this context. Due to the similarity of the problems, we think it is not unlikely that maybe techniques such as Color-Coding or combining solutions via convolution could be extended to ILPs as well.

Chapter 5

Scheduling Compressed Monotone Moldable Jobs using Convolution

In the moldable job scheduling problem one has to assign a set of n jobs to m machines, in order to minimize the time it takes to process all jobs. Each job is moldable, so it can be assigned not only to one but any number of the equal machines. We assume that the work of each job is monotone and that jobs can be placed non-contiguously. In this work we present a $(\frac{3}{2}+\epsilon)$-approximation algorithm with a worst-case runtime of $O(n\log^2(\frac{1}{\epsilon}+\frac{\log(\epsilon m)}{\epsilon})+\frac{n}{\epsilon}\log(\frac{1}{\epsilon})\log(\epsilon m))$ when $m \leq 16n$. This is an improvement over the best known algorithm of the same quality by a factor of $\frac{1}{\epsilon}$ and several logarithmic dependencies. We complement this result with an improved FPTAS with running time $O(n\log^2(\frac{1}{\epsilon}+\frac{\log(\epsilon m)}{\epsilon}))$ for instances with many machines $m > 8\frac{n}{\epsilon}$. This yields a $\frac{3}{2}$-approximation with runtime $O(n\log^2(\log m))$ when $m > 16n$.

We achieve these results through one new core observation: In an approximation setting one does not need to consider all m possible allotments for each job. We will show that we can reduce the number of relevant allotments for each job from m to $O(\frac{1}{\epsilon}+\frac{\log(\epsilon m)}{\epsilon})$. Using this observation immediately yields the improved FPTAS. For the other result we use a reduction to the knapsack problem first introduced by Mounié, Rapine and Trystram. We use the reduced number of machines to give a new elaborate rounding scheme and define a modified version of this knapsack instance. This in turn allows for the application of a convolution based algorithm by Axiotis and Tzamos. We further back our theoretical results through a practical implementation and compare our algorithm to the previously known best result.

5.1 Introduction

The machine scheduling problem, where one assigns jobs to machines in order to finish all jobs in a preferably short amount of time, has been a core problem of computer science. Its applications are not only limited to the usual context of executing programs on a range of processor cores but it also has many applications in the real world. For example one can view machines as workers and jobs as tasks or assignments that need to be done. In reality multiple workers can work together on a singular task to complete it quicker. This however gives rise to another layer of this problem: Before assigning a starting time to a job, we first need to decide how many machines will execute the job. The resulting problem is considered as Parallel Task Scheduling with Moldable Jobs. Our goal is to minimize the time when the last job finishes, which is called the *makespan*.

In this problem the time necessary for a job to be processed is dependent on the number of assigned machines. We further consider the setting where our jobs are not only moldable but also have monotone work. The work of a job j with k machines is defined as $w(j,k) := t(j,k) \cdot k$, which intuitively is the area of the job. We assume that this function for a fixed job j is non-decreasing in the number of machines. This assumption is natural since distributing the task on multiple machines will not reduce the actual amount of work. Instead invoking multiple machines will more likely induce a bit of overhead due to necessary communication among the machines.

Since finding an optimal solution to this problem is NP-hard [63] our goal is to present approximation algorithms. Such an algorithm has to guarantee for every instance I with optimal makespan $OPT(I)$ to find a solution with a makespan of at most $c \cdot OPT(I)$ for some multiplicative approximation ratio $c > 1$. In this section we introduce two algorithms that take an additional accuracy parameter $\epsilon > 0$ as input and guarantee solution quality based on this parameter.

The first guarantees an approximation ratio of $c_1 = 1 + \epsilon$ in time $O(n \log^2(\frac{4}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon}))$ under the additional premise that $m > 8\frac{n}{\epsilon}$. Our second algorithm achieves an approximation ratio of $c_2 = \frac{3}{2} + \epsilon$ with running time $O(n \log^2(\frac{1}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon}) + \frac{n}{\epsilon}\log(\frac{1}{\epsilon})\log(\epsilon m))$ when $16n \geq m$. If we apply the first algorithm for $\epsilon = \frac{1}{2}$ and combine both algorithms we get an efficient $(\frac{3}{2} + \epsilon)$-approximation.

We achieve our results through a new core observation: Although a job can be assigned to every possible number of machines, not all m different allotments may be relevant when looking for an approximate solution. In fact we will show that if m is large enough we can reduce the number of relevant machine allotments to $O(\frac{1}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon})$. This overall assessment is based on the concept of *compression* introduced by Jansen and Land [59].

We use the reduced number of relevant allotments to schedule moldable jobs via an instance of the knapsack problem. This approach was initially introduced by Mounié, Rapine and Trystram [76]. We give a new rounding scheme to convert moldable jobs into knapsack items to define a modified version of their knapsack instance. The number of different item sizes and profits in our instance is small. This allows for the application of a knapsack algorithm introduced by Axiotis and Tzamos [2] using convolution. Their algorithm works well on such instances and thanks to our rounding, we can even do the required pre-processing for their algorithm efficiently in linear time.

5.1.1 Problem definitions and notations

Two problems will play an important role: The first being parallel task scheduling with moldable jobs, which we will call moldable job scheduling in the following. In this problem one is given a set J of n jobs and a set M of m equal machines. We write $[l] = \{i \in \mathbb{N} \mid 1 \leq i \leq l\}$ for any $l \in \mathbb{N}$. The processing time of a job in the moldable setting is defined as a function $t : J \times [m] \to \mathbb{R}_{\geq 0}$ where $t(j,k)$ denotes the processing time of job j on k machines. We denote with $\gamma(j,d) = \min\{i \in [m] \mid t(j,i) \leq d\}$ the number of machines required for job j to achieve processing time smaller than d. If d is not achievable with m machines, we say $\gamma(j,d)$ is undefined.

For a solution of this problem we require two things: First an allotment $\alpha : J \to [m]$ and an assignment of starting times $s : J \to \mathbb{R}_{\geq 0}$. For simplicity we denote $\alpha_j := \alpha(j)$ and $s_j := s(j)$ respectively. A feasible solution must now satisfy that at any time at most m machines are in use. Denote with $U(t) := \{j \in J \mid t \in [s_j, s_j + t(j,\alpha_j)]\}$ the

jobs that are processed at time t. If at all times $t \in \mathbb{R}_{\geq 0}$ we have that $\sum_{j \in U(t)} \alpha_j \leq m$ then the schedule defined by α and s is feasible.

Finally we look to minimize the makespan of this schedule, which is the time, when the last job finishes. Given an allotment α and starting times s the makespan is defined by $\max_{j \in J}\{s_j + t(j, \alpha_j)\}$. As mentioned before the work of a job is defined as $w(j,k) = k \cdot t(j,k)$. We will work under the assumption that this work function for each job is non-decreasing. More precisely for all jobs j and $k, k' \in [m]$ with $k \leq k'$ we have $w(j,k) \leq w(j,k')$.

The second main problem we consider in this work is the knapsack problem [1], as it will be part of our algorithm to solve a knapsack instance. In the knapsack problem one is given a set of n items where each item i is identified with a profit value $p_i \in \mathbb{R}_{>0}$ and a size or weight $w_i \in \mathbb{N}$. The task is to find a maximum profit subset of these items such that the total weight does not exceed a given capacity $t \in \mathbb{N}$.

5.1.2 Related work

The moldable job scheduling problem is known to be NP-hard [39] even with monotone work functions [63]. Further there is no polynomial time approximation algorithm with a guarantee less than $\frac{3}{2}$ unless P=NP [38]. Belkhale and Banerjee gave a 2-approximation for the problem with monotony [10], which was later improved to the non-monotone case by Turek et al. [80]. Ludwig and Tiwari improved the running time further [74] and achieved a running time polylogarithmic in m, which is especially important for compact input encoding, where the length of the input is dependent on $\log m$ and not m.

Mounié et al. gave a $(\frac{3}{2} + \epsilon)$-approximate algorithm with running time $O(nm \log \frac{1}{\epsilon})$ [76]. Jansen and Land later improved this result further by giving an FPTAS for instances with many machines and complementing this with an algorithm that guarantees a ratio of $(\frac{3}{2} + \epsilon)$ with polylogarithmic dependence on m. They picked up on the idea of Mounié et al. to use a knapsack instance to find a schedule distributing jobs in two shelfs and modified the knapsack problem to solve it more efficiently. In a recent result Wu et al. [83] gave a new $\frac{3}{2}$- approximation that works in time $O(nm \log(nm))$

The Knapsack problem as a generalization from Subset Sum is another core problem of computer science that is NP-hard as well. For this problem pseudopolynomial algorithms have been considered starting with Bellmans classical dynamic programming approach in time $O(nt)$ [11]. Many new results with pseudopolynomial running times have recently been achieved in regards to various parameters such as largest item size or number of different items [41, 77, 2, 9].

One interesting connection has come up between Knapsack and the $(\max, +)$-convolution problem. In this problem one is given two sequences $(a_i)_{0 \leq i < n}, (b_i)_{0 \leq i < n}$ of length n and has to find the convolution $c = a \oplus b$ which is defined through $c_i = \max_{j \leq i}(a_j + b_{i-j})$ for all $i \in \mathbb{N}_{<n}$. This problem can be solved in quadratic time $O(n^2)$. Cygan et al. [36] conjecture that a subquadratic algorithm may not be possible and used this conjecture as a basis for many fine-grained complexity results for Knapsack and similar problems. Axiotis and Tzamos showed that with concave sequences, convolutions can be computed in linear time $O(n)$ and they used this to give a $O(Dt)$ for Knapsack where D is the number of different item sizes [2]. This approach has also been used by Polak et al. [77] in conjunction with proximity arguments from Eisenbrand Weismantel [41] to gain fast algorithms for knapsack with small item sizes .

[1] We mainly consider $0-1$ Knapsack here, though some items may appear multiple times.

5.1.3 Our results

We present a new algorithm, in particular a $(\frac{3}{2}+\epsilon)$-approximation algorithm, for any accuracy parameter $\epsilon > 0$, with a runtime polynomial in n, $\frac{1}{\epsilon}$ and in $\log m$. Since we are polynomial in $\log m$, our algorithm will be able to handle certain compact input encodings and will generally scale well into large m.

The main difficulty in moldable job scheduling is that for every job we need to choose between m different allotments and then schedule jobs efficiently. We will however show that not all m possible allotments have to be considered. Since we look for an approximate solution and we have monotone jobs, it is sufficient to only consider $O(\frac{1}{\epsilon}+\frac{\log(\epsilon m)}{\epsilon}))$ different machine counts. This immediately leads to a fully polynomial time approximation scheme (FPTAS) for instances with many machines.

Theorem 5.1.1. *Let $\epsilon > 0$. For moldable job scheduling with instances where $m > 8\frac{n}{\epsilon}$ exists a $(1+\epsilon)$-approximation that runs in time $O(n\log^2(\frac{1}{\epsilon}+\frac{\log(\epsilon m)}{\epsilon}))$.*

This result can be used for a $\frac{3}{2}$-approximation if we set $\epsilon = \frac{1}{2}$.

Corollary. *Consider moldable job scheduling on instances with $m > 16n$. There exists a $\frac{3}{2}$-approximation in time $O(n\log^2(\log m))$.*

We complement this result with an efficient $(\frac{3}{2}+\epsilon)$-approximation for the case where $m \leq 16n$. To achieve this we follow the same approach as [59, 76] and construct a knapsack instance. We will introduce a new rounding scheme for machine counts, processing times and job works and convert these modified jobs into knapsack items. The resulting knapsack instance will only have a small number of different item sizes. We then apply an algorithm introduced by Axiotis and Tzamos [2] that works well on such instances. Thanks to our rounding we will be able to do the pre-processing of their algorithm in linear time as well.

Theorem 5.1.2. *For moldable job scheduling there exists an algorithm that for instances with $m \leq 16n$ and for any $\epsilon > 0$ yields a $\frac{3}{2}+\epsilon$ approximation in time:*
$O(n\log^2(\frac{1}{\epsilon}+\frac{\log(\epsilon m)}{\epsilon})+\frac{n}{\epsilon}\log(\frac{1}{\epsilon})\log(\epsilon m))$

These two results combined make up one $(\frac{3}{2}+\epsilon)$-approximation that improves on the best known result by Jansen and Land [59] in multiple ways. For large m we manage to reduce the dependency on m even further. When m is small we improve on their running time by reducing the dependency on ϵ by a factor of $\frac{1}{\epsilon}$ and several polylogarithmic factors. We also argue that our algorithm is overall simpler compared to theirs, as we do not require to solve knapsack with compressible items in a complicated manner. Instead our algorithm merely constructs the modified knapsack instance and delegates to a simple and elegant algorithm from Axiotis and Tzamos [2].

RESULT	JANSEN & LAND [59]	THIS WORK
$1+\epsilon$, $(m > 8\frac{n}{\epsilon})$	$O(n\log(m)(\log(m)+\log(\frac{1}{\epsilon})))$	$O(n\log^2(\frac{1}{\epsilon}+\frac{\log(\epsilon m)}{\epsilon}))$
$\frac{3}{2}$, $(m > 16n)$	$O(n\log^2(m))$	$O(n\log^2(\log m))$
$\frac{3}{2}+\epsilon$, $(m \leq 16n)$	$O(\frac{n}{\epsilon^2}\log m(\frac{\log m}{\epsilon}+\log^3(\epsilon m)))$	$O(n\log^2(\frac{1}{\epsilon}+\frac{\log(\epsilon m)}{\epsilon})+\frac{n}{\epsilon}\log(\frac{1}{\epsilon})\log(\epsilon m))$

5.2 General Techniques and FPTAS for many machines

The core technique we use is the concept of *compression* introduced by Jansen and Land [59]. The general idea of compression is to reduce the number of machines a job is assigned to by some fraction. Due to monotonicity the resulting increase of processing time can then be bounded.

Lemma 5.2.1 ([59]). *Let $\rho \in (0, 1/4]$ be what we denote in the following as a compression factor. Consider now a job j and a number of machines $k \in \mathbb{N}$ with $\frac{1}{\rho} \leq k \leq m$, then we have that $t(j, \lfloor (1-\rho)k \rfloor) \leq (1+4\rho)t(j,k)$.*

The intuitive interpretation of this lemma is that if a job uses $k \geq \frac{1}{\rho}$ machines then we can free up to $\lceil \rho k \rceil$ machines and the processing time increases by a factor of 4ρ. We are going to use this lemma in the following by introducing a set of predetermined machine counts.

Definition 5.2.1. Let ρ be a compression factor and set $b := \frac{1}{\rho}$. We define $S_\rho := \lfloor \lfloor b \rfloor \rfloor \cup \{ \lfloor (1+\rho)^i b \rfloor \mid i \in [\, \lceil \log_{1+\rho}(\frac{m}{b}) \rceil \,] \} = \{1, 2, \cdots, b, \lfloor (1+\rho)b \rfloor, \lfloor (1+\rho)^2 b \rfloor, \cdots, m\}$ as the set of ρ-compressed sizes.

Note that reducing machine numbers to the next smaller size in S_ρ corresponds to a compression and processing time may only increase by a factor of at most $1+4\rho$. We assume without loss of generality that $1/\epsilon$ is integral by modifying ϵ. With this assumption $1/\rho$ is also integral.

Corollary. *Let $\epsilon \in (0,1)$ be an accuracy parameter then $\rho = \frac{\epsilon}{4}$ is a compression factor and $|S_\rho| \in O(\frac{1}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon})$. Further taking any number of machines $m' \leq m$ and reducing m' to the next smaller size in S_ρ corresponds to a compression.*

Proof. The first part follows from Theorem 5.2.1 with $\epsilon < 1$ and Definition 5.2.1. We show the second part by proving that for $m' := \lfloor (1+\rho)^{i+1} b \rfloor - 1$ the next smaller size $m_\rho = \lfloor (1+\rho)^i b \rfloor$ fulfills that $m' - m_\rho \leq \lceil \rho m' \rceil$. We have that

$$\begin{aligned} m' - m_\rho &= \lfloor (1+\rho)^{i+1} b \rfloor - 1 - \lfloor (1+\rho)^i b \rfloor \\ &\leq (1+\rho)^{i+1} b - 1 - ((1+\rho)^i b - 1) \\ &= (1+\rho)^{i+1} b - (1+\rho)^i b = \rho(1+\rho)^i b \leq \rho(m_\rho + 1) \leq \rho m' \end{aligned}$$

□

Generally our algorithms will work on the set S_ρ for $\rho = \frac{4}{\epsilon}$ and only assign machine counts in S_ρ. Note that for $m \leq \frac{4}{\epsilon}$ we work with any machine number as we have that $S_\rho = [m]$. The algorithms we present will work in a dual approximation framework.

A dual approximation framework is a classical approach for scheduling problems. The general idea is to use an approximation algorithm with constant ratio c on a given instance in order to gain a solution with makespan T. While this is only an approximation we can conclude that the makespan T^* of an optimal solution must be in the interval $[\frac{T}{c}, T]$ and we can search this space via binary search. We then consider candidates $d \in [\frac{T}{c}, T]$ as guesses for the optimal makespan.

The approximation algorithm is then complemented with an estimation algorithm, that receives an instance I and a guess for the makespan d as input. This estimation algorithm then must be able to find a schedule with a makespan of at most

$(1+\epsilon)d$ if such a schedule exists. If d was chosen too small, i.e. $(1+\epsilon)d < OPT(I)$, our algorithm can reject the value d and return false.

We continue to apply the estimation algorithm for candidates, until we find d such that the algorithm is successful for d but not for $\frac{d}{1+\epsilon}$. Note that if the algorithm fails for $\frac{d}{1+\epsilon}$ we have that $d = (1+\epsilon)\frac{d}{1+\epsilon} < OPT(I)$. Therefore the solution generated for d has a makespan of at most $(1+\epsilon)d < (1+\epsilon)OPT(I)$. Using binary search we can find such a candidate d in $O(\log \frac{1}{\epsilon})$ iterations [59].

5.2.1 Constant factor approximation

Our constant factor approximation operates in two steps: First we compute an allotment and assign each job to a number of machines. Secondly we will use list scheduling in order to schedule our now fixed parallel jobs. For the first step we use an algorithm introduced by Ludwig and Tiwari [74].

Lemma 5.2.2 ([74]). *Let there be an instance I for moldable job scheduling with n jobs and m machines. For an allotment $\alpha : J \to [m]$ we denote with*

$$\omega_\alpha := \min\left(\frac{1}{m}\sum_{j\in J} w(j,\alpha(j)), \max_{j\in J} t(j,\alpha(j))\right)$$

the trivial lower bound for any schedule that follows the allotment α. Furthermore for $S \subseteq [m]$ we denote with $\omega_S := \min_{\alpha: J\to S} \omega_\alpha$ the trivial lower bound possible for any allotment, which allots any job to a number of machines in S.

For any $S \subseteq [m]$ we can compute an allotment $\alpha : J \to S$ with $\omega_\alpha = \omega_S$ in time $O(n \log^2 |S|)$.

We apply this lemma but limit machine numbers to ρ-compressed sizes S_ρ for $\rho = \frac{\epsilon}{4}$. With that we gain an approximate value of $\omega_{[m]}$.

Lemma 5.2.3. *Given an instance I for moldable job scheduling with n jobs, m machines and accuracy $\epsilon < 1$. In time $O(n \log^2(\frac{4}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon}))$ we can compute an allotment $\alpha' : J \to [m]$ such that $\omega_{\alpha'} \leq (1+\epsilon)\omega_{[m]}$.*

Proof. Let $\rho = \frac{\epsilon}{4}, b = \frac{1}{\rho}$ and S_ρ be the set of ρ-compressed sizes by definition 5.2.1. We now use lemma 5.2.2 to compute an allotment $\alpha' : [n] \to S_\rho$ such that $\omega_{\alpha'} = \omega_{S_\rho}$ and note that the proposed running time follows from corollary 5.2 and lemma 5.2.2. It remains to show that $\omega_{\alpha'} \leq (1+\epsilon)\omega_{[m]}$.

Let α be an allotment with $\omega_\alpha = \omega_{[m]}$. We now modify this allotment by rounding its assigned number of machines down to the next value in S_ρ. To be more precise let $\alpha'' : [n] \to S_\rho; j \mapsto \max\{s \in S_\rho | s \leq \alpha(j)\}$. Note that based on the definitions and lemma 5.2.2 it follows immediately that $\omega_\alpha \leq \omega_{\alpha'} \leq \omega_{\alpha''}$. We will conclude the proof by showing that $\omega_{\alpha''} \leq (1+\epsilon)\omega_\alpha$.

We note that the rounding from α to α'' is a compression. To see that consider two consecutive item sizes $\lfloor b(1+\rho)^{(i-1)} \rfloor, \lfloor b(1+\rho)^{(i)} \rfloor$ for some i and note that:

$$\begin{aligned}
\lfloor b(1+\rho)^{(i)} \rfloor - \lfloor b(1+\rho)^{(i-1)} \rfloor &\leq b(1+\rho)^{(i)} - (b(1+\rho)^{(i-1)} - 1) \\
&= b(1+\rho)^{(i)} - b(1+\rho)^{(i-1)} + 1 \\
&= \rho b(1+\rho)^{(i-1)} + 1 \leq \rho b(1+\rho)^{(i)}
\end{aligned}$$

Since we only round a job down when $\alpha(j) < \lfloor b(1+\rho)^{(i)} \rfloor$ we get that $\alpha(j) - \alpha''(j) \leq \rho\alpha(j)$. According to lemma 5.2.1 the processing time of the job may only increase by a factor of at most $1 + 4\rho = 1 + \epsilon$. Therefore we have

$$\max_{j \in J} t(j, \alpha''(j)) \leq \max_{j \in J} \{(1+\epsilon) t(j, \alpha(j))\} = (1+\epsilon) \max_{j \in J} t(j, \alpha(j)).$$

Since the work function is monotone $\omega_{\alpha''} \leq (1+\epsilon)\omega_\alpha$ follows directly.

□

With this allotment we apply list scheduling to achieve a constant factor approximation [50]. We use this in our dual-approximation framework. In the next sections we therefore assume that we are given a makespan guess d and give the required estimation algorithms for the desired results.

Corollary. *The proposed algorithm is an approximation algorithm with a multiplicative ratio of* 4 *and requires time* $O(n \log^2(\frac{4}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon}))$.

Proof. The running time results mainly from applying lemma 5.2.3 to gain an allotment α with $\omega_\alpha \leq (1+\epsilon)\omega_{[m]}$. Applying list scheduling to our computed allotment yields a schedule with makespan $2\omega_\alpha \leq 2(1+\epsilon)\omega_{[m]} \leq 4OPT(I)$. □

5.3 FPTAS for large machine counts

In the section we consider instances with many machines $m > 8\frac{n}{\epsilon}$. Jansen and Land showed that an FPTAS can be achieved by simply scheduling all jobs j with $\gamma(j, (1+\epsilon)d)$ machines at time 0. They consider all possible number of machines for each job. We argue that it is sufficient to consider assigning a number in $S_{\frac{\epsilon}{4}}$ to achieve a similar result. We will however require another compression to make sure our solution is feasible.

Lemma 5.3.1. *Given an instance I with n jobs, $m > 8\frac{n}{\epsilon}$ machines and a target makespan d, we can in time $O(n \log(\frac{4}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon}))$ find a schedule with makespan $(1+3\epsilon)d$ if $d \geq OPT(I)$ or confirm that $d < OPT(I)$.*

Proof. Let S_ρ be the set of ρ-compressed sizes for $\rho = \frac{\epsilon}{4}$ and $b = \frac{1}{\rho}$. Let $\gamma'(j,d) := \max\{s \in S_\rho | s \leq \gamma(j,d)\}$ and denote a job as *narrow* when $\gamma'(j,d) \leq b$ or *wide* when $\gamma'(j,d) > b$. The schedule we propose results from scheduling narrow jobs with $\gamma'(j,d)$ machines and wide jobs with a compressed number of machines, that is $\lfloor (1-\rho)\gamma'(j,d) \rfloor$. We schedule all jobs at time 0 next to each other. The running time results from finding $\gamma'(j,d)$ for all jobs via binary search. Note that if $\gamma'(j,d)$ is undefined for some job, then d was chosen too small.

Every job j scheduled with $\gamma(j,d)$ machines has processing time of at most d. Rounding down the number of machines to $\gamma'(j,d)$ may increase the processing time by a factor of $1 + 4\rho$, as this process corresponds to a compression. We then apply another compression to wide jobs, which may increase the processing time again by the same factor. In total the new processing time of a job is bound by : $(1+4\rho)((1+4\rho)t(j, \gamma(j,d))) \leq (1+\epsilon)^2 d \leq (1+3\epsilon)d$.

It remains to show that our schedule uses at most m machines in total. Jansen and Land showed that $\sum_{j \in J} \gamma(j,d) \leq m + n$. We assume that $\sum_{j \in J} \gamma(j,d) > m$, since otherwise our schedule would be feasible already. Denote with J_W, J_N the set of wide and narrow jobs. We can see that that $\sum_{j \in J_N} \gamma(j,d) \leq n \cdot b = 4\frac{n}{\epsilon} < \frac{1}{2}m$ and therefore

$\sum_{j \in J_W} \gamma(j,d) > \frac{1}{2}m$. We will show that our rounding and compression procedure will free up enough machines.

Consider a wide job j and write $\gamma(j,d) = \gamma'(j,d) + r$ for some r. Since j was assigned to $\lfloor (1-\rho)\gamma'(j,d) \rfloor$ machines, the number of freed up machines is at least:

$$\begin{aligned}\gamma(j,d) - \lfloor (1-\rho)\gamma'(j,d) \rfloor &\geq \gamma'(j,d) + r - (1-\rho)\gamma'(j,d) \\ &= \rho\gamma'(j,d) + r \\ &\geq \rho(\gamma'(j,d) + r) = \rho(\gamma(j,d))\end{aligned}$$

In total we free at least $\sum_{j \in J_W} (\rho\gamma(j,d)) > \rho\frac{1}{2}m > \frac{\epsilon}{4}4\frac{n}{\epsilon} = n$ machines. Our schedule therefore uses at most $\sum_{j \in J} \gamma(j,d) - n \leq m + n - n = m$ machines. □

Note that we can apply this lemma for $\epsilon' = \frac{\epsilon}{3}$ or an even more simplified algorithm thats results by rounding down $\gamma(j,(1+\epsilon)d)$, which also allows a simple schedule with less than m machines [59]. If we use this algorithm in our dual approximation framework we achieve the desired FPTAS.

Proof of Theorem 5.1.1. We conclude for the runtime that we have to apply our dual approximation framework, meaning we apply the constant factor approximation and then for $\log(\frac{1}{\epsilon})$ makespan guesses we apply lemma 5.3.1. Combining these running times we get a time of $O(n \log^2(\frac{1}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon}))$. □

5.4 $(\frac{3}{2} + \epsilon)$-Approximation

We will now consider the goal of achieving a $\frac{3}{2} + \epsilon$ multiplicative approximation ratio. Our algorithm will operate again in the context of the dual approximation framework. Therefore we assume a makespan guess d and give an estimation algorithm. Our estimation algorithm will reduce the scheduling problem to a knapsack instance in a similar way that was initially introduced by Mounié et al. [76]. This approach was also used by Jansen and Land [59] who gave a modified version of this knapsack instance. We however propose a new simpler rounding scheme that uses ρ-compressed sizes for $\rho = \frac{4}{\epsilon}$ and further modify item profits. In that way we do not need a complicated algorithm to solve the knapsack problem, but we can apply the result from Axiotis and Tzamos [2].

At the start we split the set of jobs in small and big jobs $J = J_B(d) \cup J_S(d)$ with $J_S(d) := \{j \in J \mid t(j,1) \leq \frac{d}{2}\}$ and $J_B(d) = J \backslash J_S(d)$. Since we can add small items greedily at the end in linear time [59], we only need to schedule large jobs. We give a short run-down on the most important results in regards to the knapsack instance introduced by Mounié et al. .

Their main idea was to distribute all jobs into two shelfs with width m. The first shelf S_1 has height d and the second shelf S_2 has height $\frac{d}{2}$. If a job j was scheduled in either shelf with height $s \in \{d, \frac{d}{2}\}$ then j would be allotted to $\gamma(j,s)$ machines. In order to assign jobs to a shelf, they use the following knapsack instance:

Consider for each job $j \in J_B(d)$ an item with size $s_j(d) := \gamma(j,d)$ and profit $p_j(d) := w(j,\gamma(j,d/2)) - w(j,\gamma(j,d))$ and set the knapsack size to $t := m$. Intuitively this knapsack instance chooses a set of jobs J' to be scheduled in S_1. These jobs are chosen such that their work increase in S_2 would be large.

We will denote this problem as $KP(J_B(d), m, d)$ where the first two parameters declare the items and knapsack size and the third parameter is the target makespan,

which then determines the size and profits of the items. Given a solution $J' \subseteq J_B(d)$ we denote the total work of the resulting two-shelf schedule by $W(J', d)$ and note that:

$$\begin{aligned} W(J', d) &= \sum_{j \in J'} w(j, \gamma(j, d)) + \sum_{j \in J_B(d) \setminus J'} w(j, \gamma(j, \frac{d}{2})) \\ &= \sum_{j \in J_B(d)} w(j, \gamma(j, \frac{d}{2})) + \sum_{j \in J'} w(j, \gamma(j, d)) - \sum_{j \in J'} w(j, \gamma(j, \frac{d}{2})) \\ &= \sum_{j \in J_B(d)} w(j, \gamma(j, \frac{d}{2})) - \sum_{j \in J'} p_j(d) \end{aligned}$$

We can see that, as the knapsack profit is maximized, the total work $W(J', d)$ is minimized. We require the two following lemmas from Mounié et al. in the following. We refer to either [59, 76] for a detailed description of these results.

Lemma 5.4.1 ([76]). *If there is a schedule for makespan d, then there is a solution $J' \subseteq J_B(d)$ to the knapsack instance with $W(J', d) \leq md - W(J_S(d), d)$.*

Lemma 5.4.2 ([76]). *If there is a solution $J' \subseteq J_B(d)$ to the knapsack instance with $W(J', d) \leq md - W(J_S(d), d)$, then we can find a schedule for all jobs J with makespan $\frac{3}{2}d$ in time $O(n \log n)$.*

Based on these lemmas we can easily reject a makespan guess d if $W(J', d)$ is larger than $md - W(J_S(d), d)$. We note as well that lemma 5.4.2 can be applied if we find a solution for a higher makespan.

Corollary ([59]). *Let $d' \geq d$ and $J' \subseteq J_B(d)$ be a feasible solution of the knapsack problem $KP(J_B(d), m, d')$ with $W(J', d) \leq md' - W(J_S(d), d)$. Then we can find a schedule with makespan at most $\frac{3}{2}d'$ in time $O(n \log n)$.*

The goal is now to construct a modified knapsack instance in order to apply this corollary for $d' = (1 + 4\epsilon)d$. First of all we reduce machine counts to ρ-compressed sizes for $\rho = \frac{\epsilon}{4}$. Consider S_ρ and $b := \frac{1}{\rho}$ and let $\gamma'(j, s) := \max\{k \in S_\rho | k \leq \gamma(j, s)\}$ for any job j and $s \in \{\frac{d}{2}, d\}$.

If we were to construct the knapsack problem only with this modification, then items would have profits $\widetilde{p}_j(d) := \gamma'(j, \frac{d}{2}) t(j, \gamma'(j, \frac{d}{2})) - \gamma'(j, d) t(j, \gamma'(j, d))$. Call these profit values *intermediary profits*. In the next step we further modify these profits either by rounding these intermediary profits directly or by modifying the processing time. This however depends on the specifics of the job.

Next we consider a job *wide* in a shelf if it uses more than b machines in the respective shelf, that is if $\gamma'(j, s) \geq b$ for the respective $s \in \{\frac{d}{2}, d\}$. If a job is not wide we call it *narrow* instead, with respect to either shelf.

For jobs that are **narrow in both shelfs** we will directly modify the profits. Let j be a job with $\gamma'(j, s) < b$ for both $s \in \{\frac{d}{2}, d\}$, then we round the intermediary profit up to the next multiple of ϵd by setting $p'_j(d) := \min\{i\epsilon d \mid i\epsilon d \geq \widetilde{p}_j(d) \text{ and } i \in \mathbb{N}^*_{\leq \frac{2}{\epsilon^2}}\}$. This is well defined since the original profit in this case is bounded by $w(j, \frac{d}{2}) < b\frac{d}{2} = \frac{2}{\epsilon^2}\epsilon d$. For later arguments denote the modified work with $w'(j, \frac{d}{2}) := w(j, \frac{d}{2})$ and $w'(j, d) := w'(j, \frac{d}{2}) - p'_j(d)$.

For jobs j that are **wide in both shelfs**, that is when $\gamma'(j, \frac{d}{2}) \geq \gamma'(j, d) \geq b$, we will modify the processing time. In particular we set $t'(j, s) := \frac{1}{1+4\rho} s$ for $s \in \{\frac{d}{2}, d\}$,

which results in modified work values $w'(j,s) := t'(j,s)\gamma'(j,s)$. We then define the new profit based on the modified works as: $p'_j(d) := w'(j,\frac{d}{2}) - w'(j,d)$.

That leaves jobs that are **narrow in one shelf and wide in the other**. Consider such a job j with $\gamma'(j,\frac{d}{2}) \geq b > \gamma'(j,d)$. For the **wide** version we round again the processing time $t'(j,\frac{d}{2}) := \frac{1}{1+4\rho}\frac{d}{2}$ and obtain $w'(j,\frac{d}{2}) := t'(j,\frac{d}{2})\gamma'(j,\frac{d}{2})$. As for the **narrow** job we round down the work $w(j,\gamma'(j,d))$ to the next multiple of $i\epsilon d$. To be precise we set $w'(j,d) := \max\{i\epsilon d \mid i\epsilon d \leq w(j,\gamma'(j,d)) \text{ and } i \in \mathbb{N}_{\leq \frac{4}{\epsilon^2}}\}$. Note that the unmodified work is bounded by $w(j,\frac{d}{2}) \leq w(j,d) < bd = \frac{4}{\epsilon}d = \frac{4}{\epsilon^2}\epsilon d$. We then obtain the modified profit value $p'_j(d) = w'(j,\frac{d}{2}) - w'(j,d)$.

With these modified profits and sizes $s'_j(d) = \gamma'(j,d)$ we then solve the resulting problem $KP'(J_B(d),m,d,\rho)$ optimally to obtain an item set J'.

Lemma 5.4.3. *Let J' be a solution to $KP'(J_B(d),m,d,\rho)$ and $d' = (1+4\epsilon)d$, then with unmodified processing times and machine numbers J' is also a solution to $KP(J_B(d),m,d')$. Furthermore if there is a schedule with makespan d, we have that $W(J',d') \leq md' - W(J_S(d),d)$.*

Proof. For the first part we have to show that all jobs in J' fit into the respective knapsack when a processing time of d' or $\frac{d'}{2}$ for each shelf is allowed. Consider all jobs $j \in J'$ with $\gamma(j,d) \leq b$ and take note that these jobs have the same size in both knapsack instances, since $\gamma(j,d') \leq \gamma(j,d)$. For any of the wide jobs $j \in J'$ we have that $t(j,\gamma'(j,d)) \leq (1+4\rho)d \leq d'$ and therefore $\gamma(j,d') \leq \gamma'(j,d)$. We then get $\sum_{j\in J'}\gamma(j,d') \leq \sum_{j\in J'}\gamma'(j,d) \leq m$ since J' solves the modified knapsack instance which has capacity m.

Before we consider the total work of J' we want to make some observations from our rounding: We reduced the number of machines for each job by rounding the sizes. This will only reduce the work of each job due to monotonicity compared to the original knapsack instance by Mounié et al. . We then proceed to reduce work further for narrow jobs by at most ϵd and reduce the processing time of wide jobs by a factor $\frac{1}{1+4\rho}$.

Note that setting $t'(j,s) = \frac{1}{1+4\rho}s$ for a wide job j and shelf size s is actually reducing processing time and this can be seen through an indirect proof. Assume therefore $t(j,\gamma'(j,s)) < \frac{1}{1+4\rho}s$ and let $s_{k+1} := \gamma'(j,d)$ and let s_k be the next smaller size in S_ρ. Reducing the number of machines to s_k is a compression and we then have $t(j,s_k) \leq (1+4\rho)t(j,s_{k+1}) < s$. With this $\gamma'(j,s)$ was not chosen minimal.

In general we have that $w'(j,s) \leq w(j,s)$ and want to continue to give an upper bound on $w(j,s)$. Note that we may assume that processing times do not increase with increasing numbers of machines. Otherwise we could simply omit numbers of machines that increase processing times and always schedule on the smaller number. With this we get that $w(j,s) \leq (1+\rho)\gamma'(j,s)t(j,\gamma'(j,s))$.

Note that for jobs j in shelf 2 we only decrease the processing time if they are wide and therefore we get:

$$w(j,\frac{d}{2}) \leq (1+\rho)\gamma'(j,\frac{d}{2})(1+4\rho)t'(j,\gamma'(j,\frac{d}{2})) = (1+\rho)(1+4\rho)w'(j,\frac{d}{2}).$$

For wide jobs j in shelf 1 we do the same. However for narrow jobs of this shelf we reduce the work further by ϵd. Doing the same estimation for $w(j,d)$ that we did for $w(j,\frac{d}{2})$ and adding this increase, we can conclude that:

$$w(j,d) \leq (1+\rho)(1+4\rho)w'(j,d) + \epsilon d$$

For the second part of the statement we get from lemma 5.4.1 that there is an optimal solution J^* to $KP(J_B(d), m, d)$ with $W(J^*, d) \leq md - W(J_S(d), d)$. Further J^* is also a feasible solution for the modified knapsack problem, since our modifications only reduce item sizes. Our modified knapsack instance, similar to the original one, will maximize knapsack profits, which in turn then minimizes total work of a two-shelf schedule with modified work values. Since J' is an optimal solution of the modified instance, we have that the total modified work of J' is smaller than the modified work of J^*. To be precise we have:

$$\sum_{j\in J'} w'(j,d) + \sum_{j\in J_B(d)\setminus J'} w'(j,\frac{d}{2}) \leq \sum_{j\in J^*} w'(j,d) + \sum_{j\in J_B(d)\setminus J^*} w'(j,\frac{d}{2}).$$

We now can conclude that the total work of the two-shelf schedule implied by J' is bounded:

$$\begin{aligned}
W(J',d) &= \sum_{j\in J'} w(j,d) + \sum_{j\in J_B(d)\setminus J'} w(j,\frac{d}{2}) \\
&\leq \sum_{j\in J'} ((1+\rho)(1+4\rho)w'(j,d) + \epsilon d) + \sum_{j\in J_B(d)\setminus J'} (1+\rho)(1+4\rho)w'(j,\frac{d}{2}) \\
&\leq |J'|\epsilon d + (1+\rho)(1+4\rho)(\sum_{j\in J'} w'(j,d) + \sum_{j\in J_B(d)\setminus J'} w'(j,\frac{d}{2})) \\
&\leq |J'|\epsilon d + (1+\rho)(1+4\rho)(\sum_{j\in J^*} w'(j,d) + \sum_{j\in J_B(d)\setminus J^*} w'(j,\frac{d}{2})) \\
&\leq |J'|\epsilon d + (1+\rho)(1+4\rho)(\sum_{j\in J^*} w(j,d) + \sum_{j\in J_B(d)\setminus J^*} w(j,\frac{d}{2})) \\
&\leq m\epsilon d + (1+\rho)(1+4\rho)(md - W(J_S(d),d)) \\
&\leq (1+4\epsilon)md - W(J_S(d),d) = md' - W(J_S(d),d)
\end{aligned}$$

Lastly due to monotonicity of work we have also that $W(J', d') \leq W(J', d)$, which concludes the proof.

□

5.4.1 Solving the knapsack problems

As we already mentioned we intend to use an algorithm from Axiotis and Tzamos [2]. Their algorithm consists of two main steps. In the first step the items of the knapsack instance are partitioned into sets containing items of equal size. The knapsack problem is then solved for each item set separately and for every item size s with item set $I_s = \{i \in I \mid s_i = s\}$ a solution array R_s is generated where $R_s[t']$ denotes the maximum profit achievable for a knapsack of size $t' \leq t$ using only items with size s. Note that by the nature of this problem $R_s[t']$ will always be given by the sum of profits of the $\lfloor \frac{t'}{s} \rfloor$ items with the highest profit in I_s.

These solution arrays R_s have a special structure as $R_s[k \cdot s] = R_s[k \cdot s + s']$ for all $s' < s$ and $k \in \mathbb{N}$. Further considering the unique entries we have that $R_s[(k+1)\cdot s] - R_s[k \cdot s] \geq R_s[(k+2)\cdot s] - R_s[(k+1)\cdot s]$ for each k, since the profit of the items added decreases. We call arrays with this properties *s-step concave* as the unique entries build a concave sequence. In the second step of their algorithm they combine the solution arrays in sequential order via convolution to generate a final solution array $R = R_1 \oplus R_2 \oplus \cdots \oplus R_{[s_{max}]}$.

A very important result from Axiotis and Tzamos is that if these convolutions are done in sequential order, then one sequence will always be s-concave for some respective s. They proved in their paper that convolution with one s-step-concave sequence can be done in linear time, opposed to the best known quadratic time.

Lemma 5.4.4 ([2]). *Given any sequence A and R_h for some $h \in \mathbb{N}$, each with t entries, we can compute the convolution $A \oplus R_h$ in time $O(t)$.*

In our setting the knapsack capacity is given by $t = m$. Thanks to our rounding we only have $|S_\rho|$ different item sizes, which defines the number of convolutions we have to calculate. We however must also compute the initial solutions that consist of the highest profit items for each size. Thanks to rounding item profits we can also sort these efficiently to generate the initial solutions arrays R_h.

Lemma 5.4.5. *Given a modified knapsack instance $KP'(J_B(d), m, d, \rho)$, we can compute for all $t \leq t$ the entry $R_h[t']$ in time $O(n + m(\frac{1}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon}))$.*

Proof. Our goal is to sort items by profits and subsequently add up the highest profits to fill the arrays R_h. We will sort items based on how they were rounded:

Consider jobs j with $\gamma'(j,s) < b$ for both $s \in \{\frac{d}{2}, d\}$ and denote the number of these jobs with n_1. By scaling their profits with $\frac{1}{\epsilon}\frac{1}{d}$ we obtain profits of the form $\tilde{p}_j(d) = i$ for some $i \in \mathbb{N}_{\leq \frac{2}{\epsilon^2}}$. We can sort profits using radix sort in time $O(n_1 + \frac{1}{\epsilon})$ where we encode them using $O(1)$ digits ranging from 0 to $\frac{1}{\epsilon}$.

Consider now the n_2 jobs j with $\gamma'(j, \frac{d}{2}) \geq \gamma'(j,d) \geq b$. If we scale the profit of these items with $\frac{1+4\rho}{d}$ then we have that $\tilde{p}_j(d) = \frac{1}{2}\gamma'(j, \frac{d}{2}) - \gamma'(j,d)$. These items can be sorted by profit using bucket sort in $O(n_2 + m)$.

For the remaining n_3 of the jobs j with $\gamma'(j, \frac{d}{2}) \geq b > \gamma'(j,d)$ we have to consider the modified profits $p'_j(d) := \frac{d}{2(1+4\rho)}\gamma'(j, \frac{d}{2}) - i\epsilon d$ for some $i \in \mathbb{N}$. We scale these profits with $\frac{2(1+\epsilon)}{d\epsilon^2}$ to obtain $\tilde{p}_j(d) = \gamma'(j, \frac{d}{2})\frac{1}{\epsilon^2} - \frac{2id}{\epsilon} \leq \frac{m}{\epsilon^2}$. These items can be sorted with radix sort in time $O(n_3 + \frac{m}{\epsilon})$ by encoding profits with two digits ranging from 0 to $\frac{m}{\epsilon}$.

Putting these three steps together results in a total time of $O(n_1 + n_2 + n_3 + \frac{1}{\epsilon} + m + \frac{m}{\epsilon}) = O(n + \frac{m}{\epsilon})$. We can additionally merge the three sorted lists via merge sort in $O(n)$ and iterate through all items to fill the actual solution arrays. The number of total entries we have to fill in is at most $m(\frac{4}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon})$ since we have m entries in each array, and one array for every item size. □

Technically we only need the unique entries of these solution arrays to apply the algorithm [77]. These could effectively be calculated in time $O(n + \frac{m}{\epsilon})$ but combining all arrays will dominate the running time regardless.

Corollary. *We can compute $R_1 \oplus R_2 \oplus \cdots \oplus R_{|S_\rho|}$ in time $O(m(|S_\rho|))$.*

With this knapsack solution we can construct a schedule using corollary 5.4. We note that this final construction using the procedure from Mounié et al. [76] can be implemented in time $O(n)$ by using rounded processing times [59].

Proof of Theorem 5.1.2. We apply the dual approximation framework, which means we compute an upper bound for d in time $O(n \log^2(\frac{1}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon}))$. We end up with $\log(\frac{1}{\epsilon})$ candidates for d and construct knapsack instances for all of them.

To do so we need to identify their machine count among compressed sizes. This can be done in $O(n \log(\frac{1}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon}))$ via binary search. All further modifications to

knapsack items can be done in $O(n)$. In total for all candidates these steps take time $O(n \log^2(\frac{1}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon}))$.

Solving the resulting knapsack problem for one candidate can be done in time $O(m(\frac{1}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon})) \subseteq O(m\frac{1}{\epsilon}\log(\epsilon m))$. By applying this to all candidates and since $m \leq 16n$ we get a final running time of $O(n \log^2(\frac{1}{\epsilon} + \frac{\log(\epsilon m)}{\epsilon}) + \frac{n}{\epsilon}\log(\frac{1}{\epsilon})\log(\epsilon m))$. □

5.5 Implementation

We implemented all algorithms introduced here, along with a version of the algorithm introduced by Jansen and Land [59]. We note that we did not implement the final version of their algorithm to solve Knapsack with compressible items, as it was very intricate and complicated. Instead our implementation computes their modified knapsack instance and solves it via their proposed dynamic programming approach.

The implementations and experiments were conducted on a Raspberry Pi 4 Model B and we limited the experiment to one CPU-core as we did not use any mean of parallelization. We uploaded a version of our implementation to GitHub (https://github.com/Felioh/MoldableJobScheduling). In the following we mainly tested for the part where $m \leq 16n$ as we deem this the more relevant comparison between the two results.

5.5.1 Computational results

As for test instances we generated sets of randomized instances for moldable job scheduling. Machine numbers mainly range from 30 to 100 and jobs from 10 to 120. We tested on these instances for $\epsilon = \frac{1}{10}$ and the results can be seen in the following figures. Figures 5.1 and 5.2 show the difference of average runtime between our algorithm and the one by Jansen and Land. Note that the runtime of our algorithm is subtracted from the runtime of their algorithm. Hence we can see that our algorithm does slightly better for the analyzed number of jobs and machines and that our algorithm seems to scale better with growing numbers of machines and jobs.

In figures 5.3 through 5.5 we compare the average makespans of both algorithms to compare solution quality. In most cases that solution quality is generally quite similar but in some cases slightly better for our algorithm. We believe that our algorithm does better in regards to solution quality due to our rounding. For one our rounding of machine numbers to values in S_ρ is in its core a compression but does not fully utilize the potential introduced in lemma 5.2.1. Since we do not reduce the machine counts by the maximal possible amount, our effective error is smaller. In a similar manner are the additional modifications of knapsack items mainly catered to achieving a simple structure that also keeps the additional error small.

5.5.2 Computational Results (Graphs and Diagrams)

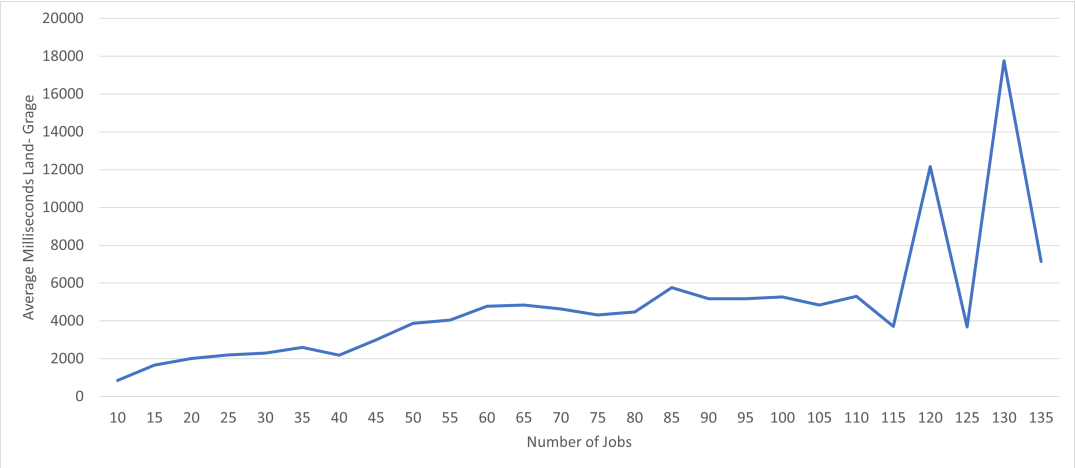


FIGURE 5.1: Average runtime difference in relation to job numbers.

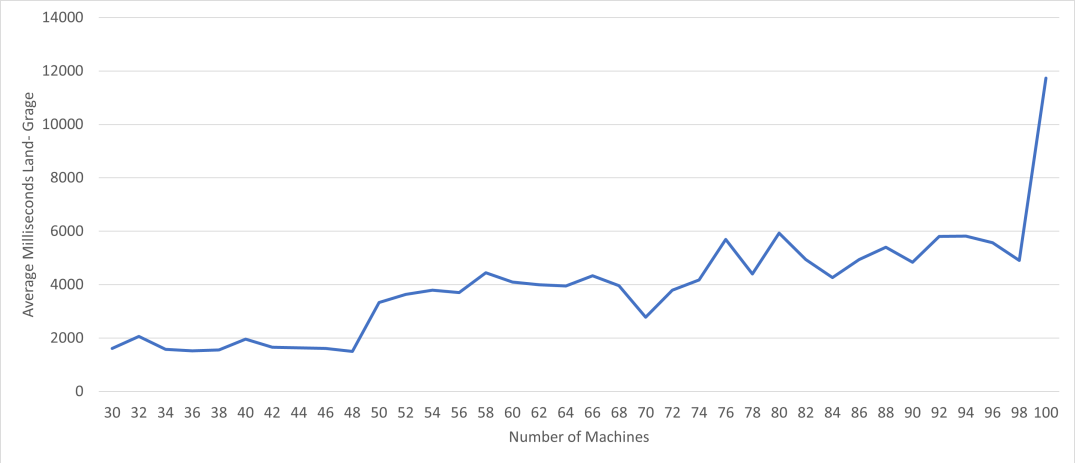


FIGURE 5.2: Average runtime difference in relation to machine numbers.

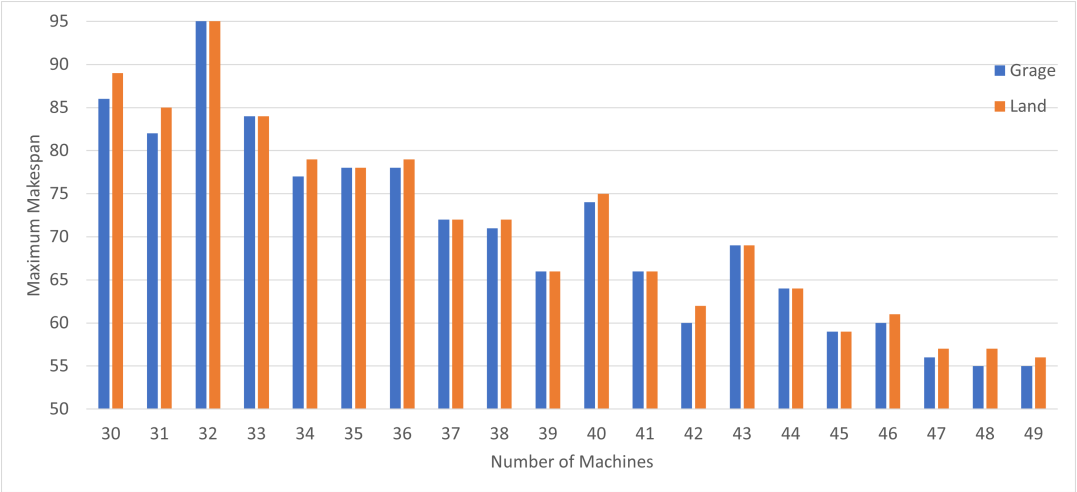


FIGURE 5.3: Average makespan comparison limited to instances with same machines.

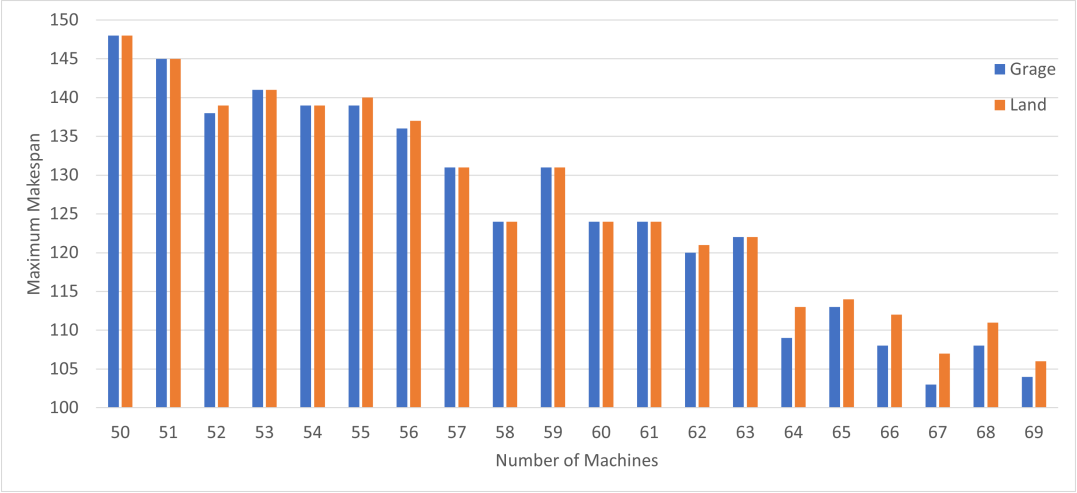


FIGURE 5.4: Average makespan comparison limited to instances with same machines.

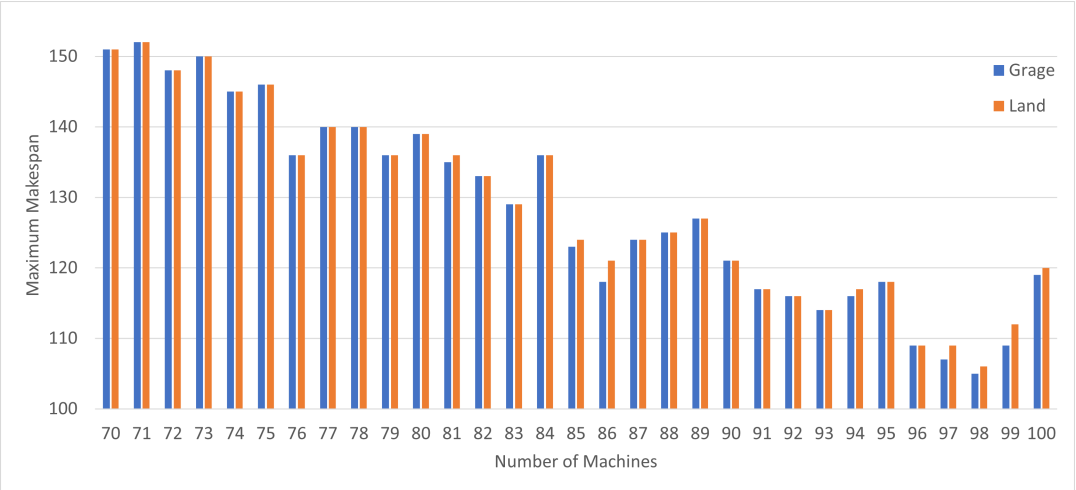


FIGURE 5.5: Average makespan comparison limited to instances with same machines.

5.6 Conclusion

In this section we presented our new $\frac{3}{2} + \epsilon$-approximation, that results from the combination of different techniques from moldable scheduling, knapsack and convolution. Our algorithm gives a theoretical improvement in terms of the known upper bound for this problem, but also proves to be faster in practice as shown by our experiments. An interesting takeaway from our result is that it is sufficient to reduce moldable scheduling to only a certain set of machine counts thanks to compression. In fact it is not necessary to regard all possible allotments, when one wants to find an approximate solution.

Bibliography

[1] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. "Proof verification and the hardness of approximation problems". In: *J. ACM* 45.3 (1998), 501–555. ISSN: 0004-5411. DOI: 10.1145/278298.278306. URL: https://doi.org/10.1145/278298.278306.

[2] Kyriakos Axiotis and Christos Tzamos. "Capacitated Dynamic Programming: Faster Knapsack and Graph Algorithms". In: *46th International Colloquium on Automata, Languages, and Programming (ICALP)*. Vol. 132. Dagstuhl, Germany, 2019, 19:1–19:13. ISBN: 978-3-95977-109-2.

[3] Brenda S. Baker. "Approximation algorithms for NP-complete problems on planar graphs". In: *J. ACM* 41.1 (1994), 153–180. ISSN: 0004-5411. DOI: 10.1145/174644.174650. URL: https://doi.org/10.1145/174644.174650.

[4] Brenda S. Baker and Jerald S. Schwarz. "Shelf Algorithms for Two-Dimensional Packing Problems". In: *SIAM J. Comput.* 12.3 (1983), pp. 508–525.

[5] János Balogh, József Békési, György Dósa, Leah Epstein, and Asaf Levin. "Lower Bounds for Several Online Variants of Bin Packing". In: *Theory of Computing Systems* (2019). ISSN: 1433-0490. DOI: 10.1007/s00224-019-09915-1. URL: https://doi.org/10.1007/s00224-019-09915-1.

[6] Nikhil Bansal, Marek Eliás, and Arindam Khan. "Improved Approximation for Vector Bin Packing". In: *Proc. SODA*. 2016, pp. 1561–1579.

[7] Nikhil Bansal and Arindam Khan. "Improved approximation algorithm for two-dimensional bin packing". In: *Proc. SODA*. 2014, pp. 13–25.

[8] Nikhil Bansal, José R. Correa, Claire Kenyon, and Maxim Sviridenko. "Bin Packing in Multiple Dimensions: Inapproximability Results and Approximation Schemes". In: *Math. Oper. Res.* 31.1 (2006), pp. 31–49.

[9] MohammadHossein Bateni, MohammadTaghi Hajiaghayi, Saeed Seddighin, and Cliff Stein. "Fast algorithms for knapsack via convolution and prediction". In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2018. Los Angeles, CA, USA: Association for Computing Machinery, 2018, 1269–1282. ISBN: 9781450355599. DOI: 10.1145/3188745.3188876. URL: https://doi.org/10.1145/3188745.3188876.

[10] Krishna P. Belkhale and Prithviraj Banerjee. "An Approximate Algorithm for the Partitionable Independent Task Scheduling Problem". In: *International Conference on Parallel Processing (ICPP)*. 1990, pp. 72–75.

[11] Richard Bellman. "Dynamic Programming". In: *Princeton University Press*. Princeton, NJ, USA, 1957.

[12] Sebastian Berndt, Klaus Jansen, and Kim-Manuel Klein. "Fully Dynamic Bin Packing Revisited". In: *Proc. APPROX-RANDOM*. 2015, pp. 135–151.

[13] Sebastian Berndt, Leah Epstein, Klaus Jansen, Asaf Levin, Marten Maack, and Lars Rohwedder. "Online Bin Covering with Limited Migration". In: *Proc. ESA (accepted)*. 2019.

[14] Sebastian Berndt, Valentin Dreismann, Kilian Grage, Klaus Jansen, and Ingmar Knof. "Robust Online Algorithms for Certain Dynamic Packing Problems". In: *Approximation and Online Algorithms - 17th International Workshop, WAOA 2019, Munich, Germany, September 12-13, 2019, Revised Selected Papers*. Vol. 11926. Lecture Notes in Computer Science. Springer, 2019, pp. 43–59.

[15] Sebastian Berndt, Valentin Dreismann, Kilian Grage, Klaus Jansen, and Ingmar Knof. "Robust Online Algorithms for Certain Dynamic Packing Problems". In: *Approximation and Online Algorithms*. Cham: Springer International Publishing, 2020, pp. 43–59. ISBN: 978-3-030-39479-0.

[16] Sebastian Berndt, Kilian Grage, Klaus Jansen, Lukas Johannsen, and Maria Kosche. "Robust Online Algorithms for Dynamic Choosing Problems". In: *Connecting with Computability - 17th Conference on Computability in Europe, CiE 2021, Virtual Event, Ghent, July 5-9, 2021, Proceedings*. Vol. 12813. Lecture Notes in Computer Science. Springer, 2021, pp. 38–49.

[17] Sujoy Bhore, Guangping Li, and Martin Nöllenburg. "An Algorithmic Study of Fully Dynamic Independent Sets for Map Labeling". In: *ACM J. Exp. Algorithmics* 27 (2022). ISSN: 1084-6654. DOI: 10.1145/3514240. URL: https://doi.org/10.1145/3514240.

[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).

[19] Nicolas Boria and Vangelis T. Paschos. "A survey on combinatorial optimization in dynamic environments". eng. In: *RAIRO - Operations Research* 45.3 (2011), pp. 241–294. URL: http://eudml.org/doc/276362.

[20] David Bremner, Timothy M. Chan, Erik D. Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, and Perouz Taslakian. "Necklaces, Convolutions, and X + Y". In: *Algorithms – ESA 2006*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 160–171. ISBN: 978-3-540-38876-0.

[21] Karl Bringmann. "A Near-Linear Pseudopolynomial Time Algorithm for Subset Sum". In: *Proceedings of the 2017 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, 2017, pp. 1073–1084. DOI: 10.1137/1.9781611974782.69. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611974782.69. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611974782.69.

[22] Karl Bringmann. *Knapsack with Small Items in Near-Quadratic Time*. 2023. arXiv: 2308.03075 [cs.DS].

[23] Karl Bringmann and Alejandro Cassis. "Faster 0-1-Knapsack via Near-Convex Min-Plus-Convolution". In: *31st Annual European Symposium on Algorithms (ESA 2023)*. Ed. by Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman. Vol. 274. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 24:1–24:16. ISBN: 978-3-95977-295-2. DOI: 10.4230/LIPIcs.ESA.2023.24. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2023.24.

[24] Alberto Caprara. "Packing 2-Dimensional Bins in Harmony". In: *FOCS*. IEEE Computer Society, 2002, pp. 490–499.

[25] Alberto Caprara, Hans Kellerer, Ulrich Pferschy, and David Pisinger. "Approximation Algorithms for Knapsack Problems with Cardinality Constraints". In: *European Journal of Operational Research* 123 (1998), p. 2000.

[26] Marcel Celaya, Stefan Kuhlmann, Joseph Paat, and Robert Weismantel. "Improving the Cook et al. Proximity Bound Given Integral Valued Constraints". In: *Integer Programming and Combinatorial Optimization*. Cham: Springer International Publishing, 2022, pp. 84–97.

[27] Timothy M. Chan and Sariel Har-Peled. "Approximation Algorithms for Maximum Independent Set of Pseudo-Disks". In: *Discrete & Computational Geometry* 48.2 (2012), pp. 373–392. ISSN: 1432-0444. DOI: 10.1007/s00454-012-9417-5. URL: https://doi.org/10.1007/s00454-012-9417-5.

[28] Timothy M. Chan and Moshe Lewenstein. "Clustered Integer 3SUM via Additive Combinatorics". In: *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*. STOC '15. Portland, Oregon, USA: Association for Computing Machinery, 2015, 31–40. ISBN: 9781450335362. DOI: 10.1145/2746539.2746568. URL: https://doi.org/10.1145/2746539.2746568.

[29] Chandra Chekuri and Sanjeev Khanna. "A Polynomial Time Approximation Scheme for the Multiple Knapsack Problem". In: *SIAM Journal on Computing* 35.3 (2005), pp. 713–728. DOI: 10.1137/S0097539700382820. eprint: https://doi.org/10.1137/S0097539700382820. URL: https://doi.org/10.1137/S0097539700382820.

[30] Lin Chen, Jiayi Lian, Yuchen Mao, and Guochuan Zhang. *Faster Algorithms for Bounded Knapsack and Bounded Subset Sum Via Fine-Grained Proximity Results*. 2023. arXiv: 2307.12582 [cs.DS].

[31] Shucheng Chi, Ran Duan, Tianle Xie, and Tianyi Zhang. "Faster Min-plus Product for Monotone Instances". In: *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2022. Rome, Italy: Association for Computing Machinery, 2022, 1529–1542. ISBN: 9781450392648. DOI: 10.1145/3519935.3520057. URL: https://doi.org/10.1145/3519935.3520057.

[32] Henrik I. Christensen, Arindam Khan, Sebastian Pokutta, and Prasad Tetali. "Approximation and online algorithms for multidimensional bin packing: A survey". In: *Computer Science Review* 24 (2017), pp. 63–79.

[33] Mark Cieliebak, Thomas Erlebach, Fabian Hennecke, Birgitta Weber, and Peter Widmayer. "Scheduling With Release Times and Deadlines on A Minimum Number of Machines". In: *Exploring New Frontiers of Theoretical Informatics*. Boston, MA: Springer US, 2004, pp. 209–222. ISBN: 978-1-4020-8141-5.

[34] 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.

[35] János Csirik and Gerhard J. Woeginger. "Shelf Algorithms for On-Line Strip Packing". In: *Inf. Process. Lett.* 63.4 (1997), pp. 171–175.

[36] Marek Cygan, Marcin Mucha, Karol Wegrzycki, and Michal Wlodarczyk. " On Problems Equivalent to (min,+)-Convolution". In: *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*. Vol. 80. (LIPIcs). 2017, 22:1–22:15. ISBN: 978-3-95977-041-5.

[37] Florian Diedrich, Rolf Harren, Klaus Jansen, Ralf Thöle, and Henning Thomas. "Approximation Algorithms for 3D Orthogonal Knapsack". In: *Journal of Computer Science and Technology* 23.5 (2008), pp. 749–762. ISSN: 1860-4749. DOI: 10.1007/s11390-008-9170-7. URL: https://doi.org/10.1007/s11390-008-9170-7.

[38] Maciej Drozdowski. "On the complexity of multiprocessor task scheduling". In: *Bulletin of The Polish Academy of Sciences-technical Sciences* 43 (1995), pp. 381–392.

[39] Jianzhong Du and Joseph Y.-T. Leung. "Complexity of Scheduling Parallel Task Systems". In: *SIAM Journal on Discrete Mathematics* 2.4 (1989), pp. 473–487.

[40] Franziska Eberle, Nicole Megow, Lukas Nölke, Bertrand Simon, and Andreas Wiese. "Fully Dynamic Algorithms for Knapsack Problems with Polylogarithmic Update Time". In: *41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2021)*. Vol. 213. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 18:1–18:17. ISBN: 978-3-95977-215-0. DOI: 10.4230/LIPIcs.FSTTCS.2021.18. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.FSTTCS.2021.18.

[41] Friedrich Eisenbrand and Robert Weismantel. "Proximity Results and Faster Algorithms for Integer Programming Using the Steinitz Lemma". In: *ACM Trans. Algorithms* 16.1 (2019). ISSN: 1549-6325.

[42] Leah Epstein and Asaf Levin. "A robust APTAS for the classical bin packing problem". In: *Math. Program.* 119.1 (2009), pp. 33–49.

[43] Leah Epstein and Asaf Levin. "Robust Algorithms for Preemptive Scheduling". In: *Algorithmica* 69.1 (2014), pp. 26–57.

[44] Leah Epstein and Asaf Levin. "Robust Approximation Schemes for Cube Packing". In: *SIAM J. Optim.* 23.2 (2013), pp. 1310–1343.

[45] Thomas Erlebach and Klaus Jansen. "The Maximum Edge-Disjoint Paths Problem in Bidirected Trees". In: *SIAM Journal on Discrete Mathematics* 14.3 (2001), pp. 326–355. DOI: 10.1137/S0895480199361259. eprint: https://doi.org/10.1137/S0895480199361259. URL: https://doi.org/10.1137/S0895480199361259.

[46] Thomas Erlebach, Klaus Jansen, and Eike Seidel. "Polynomial-Time Approximation Schemes for Geometric Intersection Graphs". In: *SIAM Journal on Computing* 34.6 (2005), pp. 1302–1323. DOI: 10.1137/S0097539702402676. eprint: https://doi.org/10.1137/S0097539702402676. URL: https://doi.org/10.1137/S0097539702402676.

[47] Björn Feldkord, Matthias Feldotto, Anupam Gupta, Guru Guruganesh, Amit Kumar, Sören Riechers, and David Wajc. "Fully-Dynamic Bin Packing with Little Repacking". In: *Proc. ICALP*. 2018, 51:1–51:24.

[48] Waldo Gálvez, José A. Soto, and José Verschae. "Symmetry Exploitation for Online Machine Covering with Bounded Migration". In: *Proc. ESA*. 2018, 32:1–32:14.

[49] Waldo Gálvez, Fabrizio Grandoni, Salvatore Ingala, Sandy Heydrich, Arindam Khan, and Andreas Wiese. "Approximating Geometric Knapsack via L-packings". In: *ACM Trans. Algorithms* 17.4 (2021). ISSN: 1549-6325. DOI: 10.1145/3473713. URL: https://doi.org/10.1145/3473713.

[50] M. R. Garey and Ronald L. Graham. "Bounds for Multiprocessor Scheduling with Resource Constraints". In: *SIAM J. Comput.* 4 (1975), pp. 187–200.

[51] Kilian Grage and Klaus Jansen. *Convolution and Knapsack in Higher Dimensions*. 2024. arXiv: 2403.16117 [cs.DS].

[52] Kilian Grage, Klaus Jansen, and Felix Ohnesorge. "Improved Algorithms for Monotone Moldable Job Scheduling Using Compression and Convolution". In: *Euro-Par 2023: Parallel Processing - 29th International Conference on Parallel and Distributed Computing, Limassol, Cyprus, August 28 - September 1, 2023, Proceedings*. Vol. 14100. Lecture Notes in Computer Science. Springer, 2023, pp. 503–517.

[53] Fabrizio Grandoni, Stefan Kratsch, and Andreas Wiese. "Parameterized Approximation Schemes for Independent Set of Rectangles and Geometric Knapsack". In: *27th Annual European Symposium on Algorithms (ESA 2019)*. Vol. 144. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 53:1–53:16. ISBN: 978-3-95977-124-5. DOI: 10.4230/LIPIcs.ESA.2019.53. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2019.53.

[54] D. V. Gribanov, I. A. Shumilov, and D. S. Malyshev. *Structured* $(\min, +)$*-Convolution And Its Applications For The Shortest Vector, Closest Vector, and Separable Nonlinear Knapsack Problems*. 2022. arXiv: 2209.04812 [cs.CC].

[55] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. "Online and dynamic algorithms for set cover". In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2017. Montreal, Canada: Association for Computing Machinery, 2017, 537–550. ISBN: 9781450345286. DOI: 10.1145/3055399.3055493. URL: https://doi.org/10.1145/3055399.3055493.

[56] Rolf Harren. "Approximation algorithms for orthogonal packing problems for hypercubes". In: *Theor. Comput. Sci.* 410.44 (2009), pp. 4504–4532. DOI: 10.1016/j.tcs.2009.07.030. URL: https://doi.org/10.1016/j.tcs.2009.07.030.

[57] Monika Henzinger. "The State of the Art in Dynamic Graph Algorithms". In: *SOFSEM 2018: Theory and Practice of Computer Science*. Cham: Springer International Publishing, 2018, pp. 40–44. ISBN: 978-3-319-73117-9.

[58] Monika Henzinger, Stefan Neumann, and Andreas Wiese. "Dynamic Approximate Maximum Independent Set of Intervals, Hypercubes and Hyperrectangles". In: *36th International Symposium on Computational Geometry (SoCG 2020)*. Vol. 164. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, 51:1–51:14. ISBN: 978-3-95977-143-6. DOI: 10.4230/LIPIcs.SoCG.2020.51. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.SoCG.2020.51.

[59] K. Jansen and F. Land. "Scheduling Monotone Moldable Jobs in Linear Time". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2018, pp. 172–181.

[60] Klaus Jansen. "A Fast Approximation Scheme for the Multiple Knapsack Problem". In: *SOFSEM 2012: Theory and Practice of Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 313–324. ISBN: 978-3-642-27660-6.

[61] Klaus Jansen. "Parameterized Approximation Scheme for the Multiple Knapsack Problem". In: *SIAM Journal on Computing* 39.4 (2010), pp. 1392–1412. DOI: 10.1137/080731207. eprint: https://doi.org/10.1137/080731207. URL: https://doi.org/10.1137/080731207.

[62] Klaus Jansen and Kim-Manuel Klein. "A Robust AFPTAS for Online Bin Packing with Polynomial Migration," in: *Proc. ICALP*. 2013, pp. 589–600.

[63] Klaus Jansen, Felix Land, and Kati Land. *Bounding the Running Time of Algorithms for Scheduling and Packing Problems*. Vol. 1302. Bericht des Instituts für Informatik. 2013.

[64] Klaus Jansen and Rob van Stee. "On strip packing With rotations". In: *Proc. STOC*. 2005, pp. 755–761.

[65] Klaus Jansen, Kim-Manuel Klein, Maria Kosche, and Leon Ladewig. "Online Strip Packing with Polynomial Migration". In: *Proc. APPROX-RANDOM*. 2017, 13:1–13:18.

[66] Ce Jin. *0-1 Knapsack in Nearly Quadratic Time*. 2023. arXiv: 2308.04093 [cs.DS].

[67] Ce Jin. "An Improved FPTAS for 0-1 Knapsack". In: *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Vol. 132. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 76:1–76:14. ISBN: 978-3-95977-109-2. DOI: 10.4230/LIPIcs.ICALP.2019.76. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2019.76.

[68] Narendra Karmarkar and Richard M Karp. "An efficient approximation scheme for the one-dimensional bin-packing problem". In: *Proc. FOCS*. 1982, pp. 312–320.

[69] Hans Kellerer. "A Polynomial Time Approximation Scheme for the Multiple Knapsack Problem". In: *Randomization, Approximation, and Combinatorial Optimization. Algorithms and Techniques*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 51–62. ISBN: 978-3-540-48413-4.

[70] Claire Kenyon and Eric Rémila. "A Near-Optimal Solution to a Two-Dimensional Cutting Stock Problem". In: *Math. Oper. Res.* 25.4 (2000), pp. 645–656.

[71] Marvin Künnemann, Ramamohan Paturi, and Stefan Schneider. "On the Fine-Grained Complexity of One-Dimensional Dynamic Programming". In: *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*. Vol. 80. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 21:1–21:15. DOI: 10.4230/LIPICS.ICALP.2017.21. URL: https://doi.org/10.4230/LIPIcs.ICALP.2017.21.

[72] Jon Lee, Joseph Paat, Ingo Stallknecht, and Luze Xu. "Improving Proximity Bounds Using Sparsity". In: *Combinatorial Optimization*. Cham: Springer International Publishing, 2020, pp. 115–127.

[73] Jakub Łącki, Jakub Oćwieja, Marcin Pilipczuk, Piotr Sankowski, and Anna Zych. "The Power of Dynamic Distance Oracles: Efficient Dynamic Algorithms for the Steiner Tree". In: *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*. STOC '15. Portland, Oregon, USA: Association for Computing Machinery, 2015, 11–20. ISBN: 9781450335362. DOI: 10.1145/2746539.2746615. URL: https://doi.org/10.1145/2746539.2746615.

[74] Walter Ludwig and Prasoon Tiwari. "Scheduling Malleable and Nonmalleable Parallel Tasks". In: *Proc. of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '94. Arlington, Virginia, USA: Society for Industrial and Applied Mathematics, 1994, 167–176. ISBN: 0898713293.

[75] Arturo Merino and Andreas Wiese. "On the Two-Dimensional Knapsack Problem for Convex Polygons". In: *ACM Trans. Algorithms* 20.2 (2024). ISSN: 1549-6325. DOI: 10.1145/3644390. URL: https://doi.org/10.1145/3644390.

[76] Grégory Mounié, Christophe Rapine, and Denis Trystram. "A 3/2-Dual Approximation for Scheduling Independant Monotonic Malleable Tasks". In: *SIAM J. Comput.* 37 (2007), pp. 401–412.

[77] Adam Polak, Lars Rohwedder, and Karol Węgrzycki. "Knapsack and Subset Sum with Small Items". In: *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*. Vol. 198. 2021, 106:1–106:19. ISBN: 978-3-95977-195-5.

[78] Peter Sanders, Naveen Sivadasan, and Martin Skutella. "Online Scheduling with Bounded Migration". In: *Math. Oper. Res.* 34.2 (2009), pp. 481–498.

[79] Martin Skutella and José Verschae. "Robust Polynomial-Time Approximation Schemes for Parallel Machine Scheduling with Job Arrivals and Departures". In: *Math. Oper. Res.* 41.3 (2016), pp. 991–1021.

[80] John Turek, Joel L. Wolf, and Philip S. Yu. "Approximate Algorithms Scheduling Parallelizable Tasks". In: *Proc. of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '92. San Diego, California, USA: Association for Computing Machinery, 1992, 323–332. ISBN: 089791483X.

[81] André Van Vliet. *Lower and Upper Bounds for On-line Bin Packing and Scheduling Heuristics: Onder-en Bovengrenzen Voor On-line Bin Packing en Scheduling Heuristieken*. PhD Thesis, 1995.

[82] Ryan Williams. "Faster All-Pairs Shortest Paths via Circuit Complexity". In: *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*. STOC '14. New York, New York: Association for Computing Machinery, 2014, 664–673. ISBN: 9781450327107. DOI: 10.1145/2591796.2591811. URL: https://doi.org/10.1145/2591796.2591811.

[83] Fangfang Wu, Xiandong Zhang, and Bo Chen. "An improved approximation algorithm for scheduling monotonic moldable tasks". In: *European Journal of Operational Research* 306.2 (2023), pp. 567–578. ISSN: 0377-2217.

[84] Andrew Chi-Chih Yao. "New algorithms for bin packing". In: *J. ACM* 27.2 (1980), pp. 207–227.

Eidesstattliche Erklärung

Hiermit erkläre ich, Kilian GRAGE, dass die vorliegende Thesis mit dem Titel "On Algorithms for Multidimensional Packing Problems and on the Complexity of higher dimensional Knapsack" , abgesehen von der Beratung von meinem Betreuer, Professor Dr. Klaus JANSEN, nach Inhalt und Form meine eigene Arbeit ist und nur mit den angegebenen Hilfsmitteln verfasst wurde. Ich bestätige weiterhin:

- Dass die Arbeit unter Einhaltung der Regeln guter wissenschaftlicher Praxis der Deutschen Forschunsgemeinschaft entstanden ist.
- Dass mir noch kein akademischer Grad entzogen wurde.

Weiterhin wurde kein Teil dieser Dissertation in einem anderen Prüfungsverfahren vorgelegt. Teile dieser Dissertation wurden bereits in wissenschaftlichen Veröffentlichungen veröffentlicht, wie in jeweils Sektion 1.1 angegeben. Weiterhin sind die Inhalte aus Kapitel 4 zum Zeitpunkt der Einreichung dieser Dissertation, beim European Symposium on Algorithms (ESA) als Teil der ALGO 2024 zur Veröffentlichung eingereicht. Ob diese Veröffentlichung angenommen wird, steht zum Zeitpunkt der Einreichung der Thesis noch aus und kann sich daher zu späterem Zeitpunkt ändern.

Unterschrift:

Datum:
